Scalable and Elastic Realtime Click Stream Analysis Using StreamMine3G

André Martin*, Andrey Brito# and Christof Fetzer*
*Technische Universität Dresden - Dresden, Germany
#Universidade Federal de Campina Grande - Campina Grande, Brazil
andre.martin@tu-dresden.de, andrey@dsc.ufcg.edu.br, christof.fetzer@tu-dresden.de

ABSTRACT

Click stream analysis is a common approach for analyzing customer behavior during the navigation through e-commerce or social network sites. Performing such an analysis in real-time opens up new business opportunities as well as increases revenues as recommendations can be generated on the fly making a previously unknown product to the potential customer attractive.

As click streams are highly fluctuating as well as must be processed in real time, there is a high demand for Event-Stream-Processing (ESP) engines that are (1) horizontally as well as vertically scalable, (2) elastic in order to cope with the fluctuation in the data stream, and (3) provide efficient state management mechanisms in order to drive such kind of analysis. However, the majority of the nowadays ESP engines such as Apache S4 or Storm provide neither explicit state management nor techniques for elastic scaling.

In this paper, we present StreamMine3G, a scalable and elastic ESP engine which provides state management out of the box, scales with the number of nodes as well as cores and improves performance due to a novel delegation mechanisms lowering contention on state as well as network links caused by fluctuations and temporary imbalances in the data streams.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Performance—*scalability, elasticity*

Keywords

ESP, Click stream analysis, Scalability, Migration, State management

1. INTRODUCTION

Click stream analysis is a key approach for making businesses in the e-commerce market competitive. For instance, such analysis enables vendors to gain insight about its customer's behavior and preferences. Vendors may then aggregate behavioral information to attract attention from users to offers that have attracted similarly behaved users. As a typical example, Amazon tracks products a customer has bought and recommends such products to other customers sharing similar interests. This is done through the comparison of both shopping carts and items visited during the navigation.

In fact, click stream analysis is not limited to the e-commerce market. The same approach enables practically any company offering a service on the Internet to continuously improve its offerings. Facebook, a popular social network uses click stream analysis to asses the perception of recently launched features and products.

Traditionally, such data analysis tasks are done combining technologies such as cloud computing and the MapReduce [12] programming model. This combination is both cost efficient and scalable (no need for business to keep clusters that are only sporadically used).

Hadoop [4], the open-source implementation of MapReduce, has become one of the most popular data processing platforms as it handles both: node failures (which is increasingly more important as the amount of data and hence the number of needed processing nodes constantly grow) and scalability (both across nodes and processing cores). Nevertheless, MapReduce targets batch processing tasks and neither supports real time processing nor quick reaction to relevant situations.

In contrast to batch processing, Event Stream Processing (ESP) is a set of techniques that specifically targets application scenarios requiring real time analytics. By using ESP techniques, businesses can react in real time to relevant changes in user behavior. Unfortunately, real time analytics impose new challenges with respect to scalability of the system. Whereas the amount of data is known a priori in a batch processing system such as Hadoop, ESP systems must be able to *elastically* scale with a continuously fluctuating workload. Especially if workloads come from sources such as the Internet, where users' behavior is very dynamic, elasticity becomes a critical.

A common type of analysis in this domain is the pattern detection over streams of user clicks, i.e., a sequence of HTTP requests. In this case, the set of pages visited by an user is analyzed and the frequency of access patterns is monitored. Although apparently simple, finding patterns in real-time data is a challenging task: First, the size of the state needed during such computations grows exponentially

DEBS'14, May 26-29, 2014, MUMBAI, India.

http://dx.doi.org/10.1145/2611286.2611304.

with both the time horizon considered (as more potential candidates need to be computed) and the number of users. Second, this size is not know a priori and may change very quickly.

However, the majority of nowadays ESP systems such as Apache S4 [20] and Storm [6] do neither support explicit state management nor elasticity[1]. Instead state must be managed by the programmer externally using either some key-value store or databases which achieves only limited scalability. Furthermore, those ESP systems do not support the migration of operators during runtime in order to cope with overload situations due to a sudden increase in the data rate. Such situations can eventually degrade performance to unacceptable levels.

In this paper we present StreamMine3G, an ESP system which aids users to perform real-time pattern detection through explicit state management. StreamMine3G supports operator migration which allows an agile adjustment of computational resources accommodating the fluctuation in the data stream. Through our novel delegation mechanism, we are furthermore able to improve throughput by efficiently coping with natural imbalances in the workload lowering contention during concurrent state access as well as network links during event dissemination.

The rest of the paper is structured as follows: In Section 2 we introduce the programming model, the architecture, and major concepts used in StreamMine3G, followed by a brief description of the state management (Section 3) as well elasticity support (Section 4). In Section 5 we present techniques used in StreamMine3G that enable both vertical and horizontal scalability of applications such as click stream analysis. We then present in Section 6 an evaluation of our system using click stream analysis, and conclude the paper with an overview of related work in Section 7 and summarize our contribution in Section 8.

2. STREAMMINE3G – PROGRAMMING MODEL AND ARCHITECTURE

StreamMine3G is an elastic ESP system offering a MapReduce-like interface, hence, users familiar with MapReduce can easily adapt their existing applications to StreamMine3G. However, contrary to MapReduce, StreamMine3G targets ESP applications, which require results to be continuously produced with the arrival of new pieces of data, here named *events*. The adaptation of data processing operations to the ESP model is often done through the usage of windows. Windows contain a number of events that is either explicitly stated or indirectly specified by considering arrivals within a time interval. Implementing the windows require that operators keep state between events. The introduction of windows and state breaks the strict phasing of MapReduce and, hence, enables real time analytics [10].

In order to implement stateful operators, users of StreamMine3G are provided with a reference to a state object in addition to the incoming event. The state object is initialized via a call back upon instantiation of the operator.

Operators in StreamMine3G can be either written in C++ or in Java through a provided JNI operator wrapper. Although the JNI wrapper imposes a performance overhead through (un)marshaling of variables, it frees StreamMine3G users from introducing unwanted memory leaks and provides a smooth integration of a wide range of existing libraries and tools.

In StreamMine3G, events traverse an acyclic graph of operators that compose a query. Each operator can receive and emit events from and to multiple upstream and downstream operators, respectively. In order to achieve horizontal scalability (i.e., scalability with increasing number of nodes), operators in StreamMine3G are partitioned in a similar fashion to MapReduce. Thus, the user is provided with a partitioner interface in addition to the operator interface. An operator partition in StreamMine3G is called a *slice* which is deployed on a StreamMine3G cluster through a central component named manager. As in MapReduce, the default partitioner is adequate to most common cases.

A StreamMine3G cluster typically consists of a set of physical or virtual machines, each running a single StreamMine3G process. Nevertheless, each process hosts an arbitrary number of slices, i.e., operator partitions. The manager component in StreamMine3G is responsible for deploying slices across a StreamMine3G cluster, and taking actions such as migrating slices if nodes are overloaded or underutilized.

In addition to the operator and partitioner interface, StreamMine3G offers a third interface to its users in order to implement a custom manager component if desired. The manager interface consists of API calls to create, deploy and wire operators for the creation of topologies, and call backs for performance monitoring of individual nodes within a StreamMine3G cluster as well as their slices running on the nodes. The performance metrics include CPU, memory, and network utilization of the machine where the StreamMine3G process is currently running on, as well as throughput and state size of each of the slices deployed on the specific node. Such information can be used by the manager in order to detect overloaded nodes and take appropriate actions such as the migration of slices to less loaded nodes and the acquisition of new resources if the current pool of resources is about to exhaust.

Contrary to open source ESP systems such as Apache S4 and Storm, topologies are not fixed and can be modified during runtime in StreamMine3G. Hence, operators can be arbitrarily wired or completely be replaced during runtime which allows the possibility of *hot-swapping* of operator code if desired. In order to achieve this, StreamMine3G uses a topic based publish subscribe mechanism internally in order to route events to their appropriate downstream operators. Every time an operator slice is deployed on the cluster, the routing tables of the affected operators are updated appropriately in order to deliver events to the correct consumers.

For load balancing and elastic data processing, the manager API provides additionally calls for the migration of operators between nodes. Operator migration allows the expansion and the shrinking of a StreamMine3G cluster as new nodes (spare capacity) can be added at any time in order to balance load evenly. On the other hand, if nodes are underused, slices with lower utilization can be collocated and idle nodes released through this mechanism. This property allows StreamMine3G to cope with arbitrary workloads and the operation in cloud environments such as Amazon EC2 [1] where customers are billed on an hourly bases for the computing resources they use.

[1]Storm offers a so-called re-balancing feature which allows to re-balance work after adding more processing nodes to a storm cluster - re-balancing must be triggered manually.

In order to give StreamMine3G users an easy head start with the technology, StreamMine3G is equipped with a default manager implementation which provides already basic functionality such as deploying a given topology across a StreamMine3G cluster.

In addition to horizontal scalability, StreamMine3G can fully utilize the computational power of current and future many-core machines. The MapReduce-like state partitioning and the hosting of multiple slices in a single StreamMine3G node enables efficient use of multi-core, heterogeneous machines.

This (vertical) scalability is achieved through a thread pool that can be expanded and shrunk during runtime, similar to the StreamMine3G cluster itself. The dynamic-sized thread pool is also suitable to newer cloud environments that allow for dynamic reconfiguration of the underlying virtual machines, i.e, adding or removing virtual cores to the VM in runtime. With this functionally, we address forthcoming cloud models that enable dynamic scalability in a hybrid fashion: vertical scaling for short-term bursts and horizontal scaling for medium and long term workload changes.

2.1 Fault Tolerance

Fault Tolerant Event Processing.

As the number of machines increase, failures get more frequent. Furthermore, ESP applications typically run continuously for long periods. In order to cope with failures, StreamMine3G supports passive as well active replication:

Passive replication is achieved through periodic checkpointing of the operator state on either stable storage or directly in memory to a non-active deployed replica slice (i.e., passive standby). If the user opts for stable storage, checkpoints can be either written to a local filesystem path which can be mapped through fuse etc. to some fault tolerant distributed file system such as the hadoop file system (HDFS [4]), or implement his own adapter in order to persist raw checkpoint data to storage services such as Amazon S3 [1] if desired. To avoid event loss between two consecutive checkpoints, event's *in-flight* are kept in an in-memory log at upstream operators which is included in checkpoints for persistence as well. The log is periodically pruned through acknowledgment messages sent from downstream operators after a successful checkpoint through the sweeping checkpointing algorithm as discussed in [13].

In order to achieve replay-ability of events needed to recover to a state that is consistent with previously produced results (precise recovery), StreamMine3G processes events deterministically as proposed in [18]. If the user opts for deterministic processing, events are ordered based on their timestamps prior to processing. Through the ordering of events, multiple input streams are merged deterministically. Besides providing precise recovery for passive replication, such an ordering also ensures consistency across slice replicas when using active replication [18].

Active replication in StreamMine3G is achieved through the deployment of multiple identical operator partitions (slices) across the cluster. As events are processed deterministically within the operators, and events are shipped in FIFO order via TCP, downstream operators can correctly discard duplicates based on the event's timestamp. The usage of active replication comes with two advantages: (1) it transparently masks permanent failures (e.g., node crashes); (2) it increases stability of processing latencies; and (3), it increase the pool of nodes that can be used for load balancing and, thus, reduces adaptation time during operator migration.

Fault Tolerant Manager.

Although recently evolved ESP systems such as Storm and Apache S4 provide fault tolerance in general, only little attention has been paid to the manager component of such systems: For example, the manager node named *nimbus* in Storm [6] presents currently the single point of failure of the system. Although topologies deployed on a Storm cluster continue to run after the crash of nimbus, a full restart of a storm cluster is needed in order to re-gain control. Contrary to those approaches, StreamMine3G provides fault tolerance at two levels: At the data processing layer as well as the cluster coordination layer for the manager component.

In order to provide fault tolerance for the manager component, StreamMine3G uses zookeeper [16] in order to store its cluster configuration in a reliable way. Zookeeper is a widely used coordination service that provides reliability through replication. As the cluster configuration is not only kept in the manager node's memory, the component can easily recovery from failures by reading the latest configuration from zookeeper and continue its operation. Similar to the extended operator interface for explicit state management as described in Section 3, user state of custom written manager implementations are also persisted transparently through zookeeper in order to provide a consistent recovery for the user's manager code.

The fault tolerance mechanism of the manager component comes with the following advantages: (1) The entire StreamMine3G cluster is resilient, and (2) the implementation of the manager code can be replaced during run time in a *hot-swapping* fashion similar as operator code without having to tear down any nodes of the cluster.

For failure detection, we use zookeeper's heart beat mechanism. If a node does not respond within an adjustable timeout (by default $5secs$), it is marked as dead and will be taken out of the set of alive nodes from the StreamMine3G cluster. Peer nodes are notified about the lost node through zookeeper's provided watcher mechanism. If the node hosting the manager node has crashed, a new node will automatically be elected (through zookeeper) as a new leader running the manager code. The manager is hereafter resumed through the previously checkpointed state in zookeeper.

3. STATEMANAGEMENT SUPPORT

StreamMine3G provides developers with an explicit state management support in order to perform arbitrary data analysis that require stateful operators. In order to offer state management to the user, we have extended the operator interface by the following methods: *initState()*, *freeState()*, *serialize()* and *deserialize()*. While the first two methods are used for the initialization of the state data structures during the deployment of an operator and freeing memory whenever an operator slice is removed from a host, the latter two serve for fault tolerance and elasticity. Instead of using external tools such as databases or key-value stores for preserving long running accumulated application state, StreamMine3G users can simply rely on the integrated state persistence and recovery mechanisms by solely providing appropriate (de-)serialization code. If the user opted

to implement his operator code in C++, user can improve checkpointing and migration performance by using flat serialization free data structures as proposed in [17]. StreamMine3G uses a *writev* syscall in order to write out state that was either provided in flat or serialized form without performing expensive *memcpy* operations in background.

In order to fully support nowadays multi-core machines, StreamMine3G supports state partitioning on top of operator partitioning. State partitioning allows the concurrent processing of two consecutive events that arrived at the same operator partition if accessing disjoint sets of state which significantly improves throughput as we will show in our evaluation in Section 6.

In order to ensure consistency during state modification, StreamMine3G employs a fine grained locking scheme preventing concurrent accesses and modifications of the same memory regions leading to corrupted state and data structures. As state as well as locking is completely managed by StreamMine3G, developers are freed from the burden of using mutexes in a correct fashion while still benefiting from a high degree on parallelism during execution.

4. ELASTICITY SUPPORT

In order to adapt to changing loads during runtime, ESP systems such as StreamMine3G must be capable of acquiring new resources during runtime and shifting, i.e., migrating operators from overloaded nodes to the newly acquired ones. Migration in StreamMine3G is accomplished by instantiating a replica at the spare node and removing the primary once the secondary replica is up and running. Stateless operators can be migrated quite easily this way as it only requires the instantiation of a new copy on the spare node and the redirection of the affected incoming and outgoing event stream. However, for stateful operators an additional step, the migration of the operator's state must be considered.

In order to migrate the state of an operator, StreamMine3G uses the user provided methods to fetch a serialized copy of the state. However, instead of writing the serialized state to stable storage as it would be done for fault tolerance to allow recovery from failed nodes, the state is directly sent to the peer node which has been previously selected to run the replica. Once the state arrives at the replica, the state is then de-serialized which brings the operator in ready state.

Since StreamMine3G processes events deterministically, duplicate events produced during the migration process can be easily filtered at downstream operators. This filtering mechanism allows the implementation of an interruption free migration process where no operator must be temporarily stopped in order to prevent the occurrence of duplicate events at downstream operators.

5. TECHNIQUES FOR IMPROVING SCALABILITY

Although applications with partition-able state can conceptionally fully scale within the two dimensions (horizontal and vertical), real world applications often exhibit a non-optimal behavior that limits scalability. For example, a poorly chosen partition key may overload certain operator partitions while leaving others underutilized. Similarly, if long sequences of events need to access the same state partition, concurrent access protection will limit the processing close to a single threaded execution.

In the following, we describe mechanisms employed in StreamMine3G that improve scalability and hence performance for real world application that may exhibit such a counter productive behavior such as pattern detection in the click stream analysis application, where certain patterns are likely to be much more frequent than others.

Network Batch Delegation.

In order to improve throughput, events in StreamMine3G are sent in batches to downstream operator slices. However, some operators may naturally emit a longer sequence of events to a single downstream slice. This effect results in a higher contention on that specific network batch and limits parallelism as threads are blocked until the network batch is fully sent. To avoid such a blocking, we introduced delegation for network batches: if a thread is currently busy with sending a complete batch downstream other threads can, in a non-blocking fashion, append ready-to-send events to a list which will then later be passed on to the thread currently in charge of sending events on that channel. Appending ready-to-send events to such lists (note: a separate list exists for each separate downstream channel), frees the previously blocked thread for the processing of events for idle downstream channels. The algorithm is depicted in Listing 1:

Algorithm 1 Network Batch Delegation

```
 1: function EMITEVENT(event)
 2:     mutexBatch.lock()
 3:     batch.add(event)
 4:     if batch.size() > threshold then
 5:         batchesToSend.add(batch)
 6:         batch← new batch
 7:         if mutexSendChannel.trylock() then
 8:             cntr ← 0
 9:             while batchesToSend <> empty do
10:                 if cntr < procSize then break
11:                 mutexBatch.unlock()
12:                 sendBatch(batchesToSend.pop())
13:                 mutexBatch.lock()
14:                 cntr ← cntr + 1
15:             mutexSendChannel.unlock()
16:         else if batchesToSend.size() > maxBPend then
17:             mutexBatch.unlock()
18:             mutexSendChannel.lock()
19:             mutexBatch.lock()
20:             for each batch in batchesToSend do
21:                 sendBatch(batch)
22:             mutexSendChannel.unlock()
23:             batchesToSend.clear()
24:     mutexBatch.unlock()
```

Every time an event is emitted by an operator, the event is first passed to the previously mentioned partitioner in order to determine the target partition of the downstream operator. StreamMine3G maintains a so called downstream channel object for each target partition. A downstream channel object consists of a queue where events are appended to during emission, and a list where those batches of events are appended once they are full and ready to be sent downstream.

In a first step the event is appended to the batch for the previously selected downstream channel (Line 3). Since StreamMine3G allows the hosting of several slices in a single node/process, appends to the same batch are protected through a mutex (Line 2). With every append, the size of the batch is checked. If the size exceeds a dynamically adjustable *threshold*, the batch is marked as full and appended to the list of full batches in order to being shipped over the wire later on. The dynamically adjustable *threshold* parameter allows us to flush events immediately to reduce latency if desired.

Once a batch is full and has been appended to the list of full batches (Line 4 and 5), a new empty batch is being created (Line 6), and the currently active thread attempts to acquire the sender mutex lock for the channel in order to send the batch downstream. If the acquisition was successful, hence, no other thread is currently busy with putting complete batches for the selected channel on the wire, the thread iterates over the list of full batches and puts the full batches one by one on the wire (Line 9-15). Note: A batch consists of multiple events, hence we use a *writev* to reduce the number of syscalls. Prior to this process, the mutex for appending new events to the previously created batch has been released (Line 10) in order to allow previously competing threads to append new events to the batch in a non-blocking fashion which increases substantially throughput as we will show in Section 6 in our evaluation.

In case the system experiences an imbalance in the workload for a longer period of time, the above algorithm would exhibit the following behavior: (1) the list of full batches would grow with time, hence increasing latency and memory consumption, and (2) the thread that entered the loop for sending the full batches first will not leave the processing loop until the imbalance dissolved.

In order to address the first issue, the algorithm in Listing 1 contains an implicit back pressure mechanisms which prevents an infinite growth of the full batches list. The growth of the list is bounded by the dynamically adjustable parameter $maxBPend$ (Line 16) which specifies the maximal number of batches pending to be sent over the wire. The parameter can be adjusted during runtime in order to post pone back pressure on upstream operators if needed. In case the limit has been exceeded, back pressure is enforced by keeping the batch mutex (Line 19) locked while sending the accumulated full batches (Line 20 and 21). Note: Batches are sent over the wire in a synchronous fashion hence the send call is blocking.

In order to allow a thread to leave the processing loop while sending batches, we limited the amount of batches a thread can send each time he successfully acquired the sender lock by using a simple counter mechanisms and a the limit variable $procSize$ (Line 8, 10 and 14).

State Access Delegation.

StreamMine3G uses delegation for accessing state in stateful operators. State access delegation prevents the queuing of threads if multiple events in a row must access the same piece of the state due to the nature of the implemented operation. In Figure 1, three events want to access state partition $S3$, hence blocking the processing of the remaining events in the incoming queue which need to access state partitions other than $S3$.

In order to prevent blocked threads such as in this scenario, events are enqueued in a processing queue that belongs to the thread that currently holds access to that piece of the state (i.e., the processing of those events is delegated to that thread). This approach reduces both: contention for the state and enables better use of a core's cache.

The algorithm for our state access delegation is similar to the algorithm depicted in Listing 1. However, instead of appending events to event batches for a delivery over the network, events are put in a FIFO processing queue. After queue insertion, the thread continues with the processing of the queue's head element if a lock acquisition for the state partition needed to process the event at hand was successful. The state partition is determined by a *stateAccess* method where the event is passed to prior to processing and acts as the partitioner in a similar way as it is used for routing events to downstream operator partitions.

6. EVALUATION

In the following section, we present our performance evaluation. We consider both horizontal and vertical scalability. In addition, we also evaluate our system's elasticity with a dynamic workload.

Our click stream analysis application subscribes to a stream of Apache HTTPd log entries from a cluster of web servers using piped logging [2]. The application consists of the three pipelined operators: The first operator groups HTTP requests by user ids (through session id cookies). The following operator keeps a sliding window of the requests per user id. With each update to the sliding window, an output is produced representing the identified pattern. The last operator is a *top-k*, which continuously tracks the frequency of the patterns. The frequency can then be used to trigger actions that consider the most popular access patterns in the site (or set of sites, if technologies such as third-party cookies are used for tracking).

Our experiments were executed on a 40-node cluster where each node is equipped with 2 Intel Xeon E5405 (quad core) CPUs and 8 GB of RAM running a Debian Linux 7.4 operating system with kernel 3.2.0. All nodes are connected via Gigabit Ethernet (1000BaseT full duplex). StreamMine3G is written in C++ providing a C++ as well as a Java interfaces for implementing operators. The click stream analysis application is implemented in C++ using StreamMine3G's native operator interface.

Horizontal Scalability.

For the first experiment we continuously added nodes for the pattern detection and top-k operator and increased the workload imposed on the application. The number of nodes where source operator partitions are deployed for accepting log entries from web servers is kept constant. The aggregated throughput was measured at the pattern detection operator.

As depicted in Figure 2, StreamMine3G scales almost linearly with the increasing workload and added computing resources. Note that the delegation mechanism for network batches considerably improves throughput. Each StreamMine3G node is processing around 160 thousand events per second. Through the back-pressure mechanism in our delegation mechanism, we keep the amount of pending network batches constant, hence, imposing an upper bound on end-

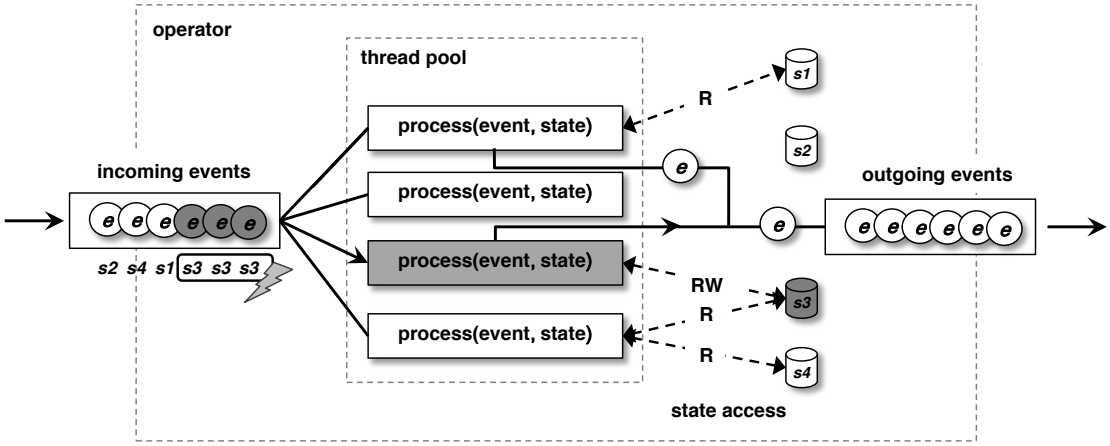

Figure 1: State access contention due to workload: Three consecutive events are trying to access state partition $S3$ limiting scalability due to sequential processing.

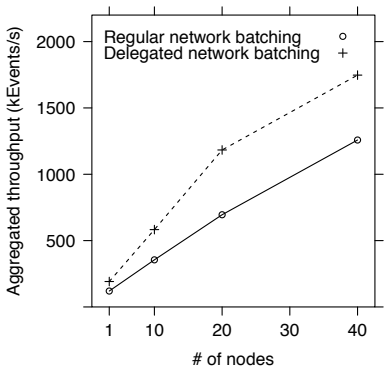

Figure 2: Horizontal scaling (aggregated throughput).

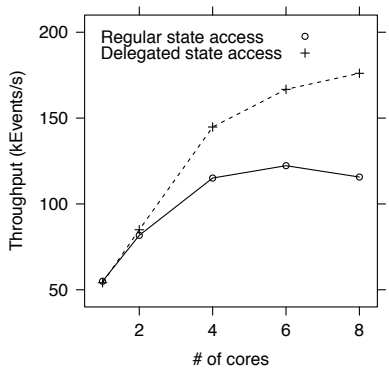

Figure 3: Vertical scaling (per node throughput).

to-end latency.

Vertical Scalability.

In the next experiment we control the size of the pool of processing threads to evaluate the throughput as the number of used cores increases. Figure 3 depicts the result of this experiment. Again, the system performance increases with the amount of available threads, i.e., the number of used cores in a node. However, if delegated state access is disabled, the throughput will decrease with more than 6 threads in use. This decrease is due to the high contention on the state which gets worse with an increasing number of threads. Although delegated state access improves throughput, the increase is not linearly as with horizontal scaling. This is due to the usage of a single event queue that exists for each operator slice. The queue is needed for ordering events if the user opted deterministic execution. However, throughput can still be improved through a more fine grained partitioning, i.e., the usage of more slices for an operator.

Elastic Scalability.

In the last experiment we varied the workload over time by generating HTTP requests in a rate that mimics a sinusoid. In order to cope with the varying workload, we use a simple elasticity manager component that constantly measures the CPU utilization of all nodes within a StreamMine3G cluster. If a node reaches an upper threshold of 80% of the CPU utilization, the manager allocates a new spare node and migrates successively slices to the newly acquired resource. If the CPU utilization drops below a lower bound threshold (40%), operator slices are moved to nodes with the least load which is still above the lower bound. In the experiment in Figure 4, we use pre-allocation as launching VMs in cloud environment such as Amazon EC2 can take up to 10 mins.

In order to assess the capabilities of our simple elasticity controller, we mimic a workload that triples within half

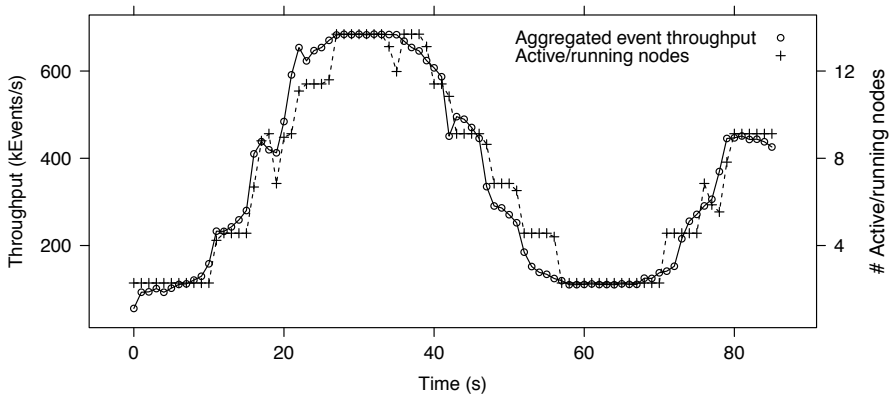

Figure 4: Elastic scaling (adapting cluster size to the workload).

a minute. As shown in Figure 4, the manager component migrates operator slices to newly-acquired spare capacity to handle the increasing workload and not suffering from service disruption due to overload. Once the workload decreases, operator slices are migrated back and co-located with other slices to free resources, releasing previously acquired nodes.

7. RELATED WORK

ESP systems have been widely studied in academia and industry for more than a decade now. One of the first academic ESP systems evolved in this area was the Borealis system [9]. Based on Borealis, Gulisano et. al. proposed StreamCloud [14], an elastic ESP systems which supports operator migration targeting cloud environments and addressing fluctuating workloads. However, StreamCloud offers only a query language based interface to its users whereas ESP systems such as Storm [6], Apache S4 [20] and StreamMine3G target MapReduce like interface where the operator itself and its state are modeled as *blackboxes*.

Although the open source system's Storm and Apache S4 allow an easy transition from batch to real time processing due to the MapReduce like interface, those systems provide only limited fault tolerance and elasticity support. While Apache S4 with it's latest 0.6 release provides simple state persistence and recovery, there is no support for operator migration. Storm provides transactional topologies guaranteeing event delivery of events, however, does not provide any state management to its users. In contrast to Apache S4, Storm provides re-balancing which allows to balance load when new nodes were added to a storm cluster during runtime. However, this type of elasticity only allows a scale up and does not consider stateful operators.

Furthermore, none of these ESP systems provide efficient multi core support resulting only in a low overall system utilization. In order to fully utilize all cores with those systems, multiple processes (Apache S4) or worker threads (Strom) must be spawned explicitely to achieve a satisfying system utilization.

StreamMine3G uses the boost.asio [3] library for network communication which allowed us to implement the network batch delegation mechanism as described in Section 5. Apache S4 and Storm are based on netty.io [5] and zeromq [8] as network substrates which allows only limited adjustments for performance improvements.

A similar variant to our network delegation mechanism has been presented by Mehul et. al. [22], however, the approach has never been empirical evaluated in terms of performance gains. Furthermore, we applied delegation approach on two levels to lower contention at network links as well as state partitions.

Fernandez et al. [11] is a work that comes closest to our contribution with regards to state management as well elasticity. The author propose explicit state management interface in order to provide fault tolerance as well elasticity to the user, however, there elasticity mechanism is limited to scale up hence cannot free resources. Our approach goes beyond as we investigated performance related problems such as locking schemes to provide consistent state modification as well as lowering contention on state due to our delegation mechanism.

8. CONCLUSION

In this paper, we presented StreamMine3G, our highly scalable and elastic ESP system. StreamMine3G supports horizontal scalability (with number of nodes) as well as vertical scalability to harness the power of nowadays modern multi-core systems.

The state management support of StreamMine3G relieves programmers from the burden of implementing their own state persistence and recovery mechanisms, and guarantees consistent state modification while offering a high degree of execution parallelism. In order to cope with fluctuating workloads, StreamMine3G supports operator migration to maximize node utilization in cloud environments such as Amazon EC2. We furthermore presented our novel delegation mechanism for minizing contention during state access as well as event dissemintion.

In order to verify and evaluate our approach, we have implemented a click stream analysis application. Our results show that our system can scale horizontally as well vertically with increasing workload, adapt dynamically to changing workloads where the system can automatically scale in (contract) or scale out (expand) to more resources if needed. Using our novel delegation mechanism, we were able to achieve 70% more throughput on a 8 core machine due to lower-

ing the contention on state. The simplicity of MapReduce programming model allows the application of the StreamMine3G approach in various domains ranging from click stream analysis such as demonstrated in this paper, deep packet inspection [21], energy prediction [19] or the implementation of an elastic content based publish subscribe system [15] on top of StreamMine3G.

Our ESP system is furthermore freely available for download and evaluation purposes [7].

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2012-2015) under grant agreement number 318809 (LEADS) and from the CAPES, DAAD and GIZ through the NoPa program (TruEGrid project, 2011-2013).

9. REFERENCES

[1] Amazon aws. `https://aws.amazon.com`, February, 15th 2014.

[2] Apache httpd piped logs. `https://httpd.apache.org/docs/2.2/logs.html#piped`, February, 15th 2014.

[3] Boost.asio. `http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html`, February, 15th 2014.

[4] hadoop. `http://hadoop.apache.org/`, February, 15th 2014.

[5] netty.io. `http://netty.io/`, February, 15th 2014.

[6] Storm. `http://storm-project.net/`, February, 15th 2014.

[7] Streammine3g. `https://streammine3g.inf.tu-dresden.de/`, February, 15th 2014.

[8] zeromq. `http://zeromq.org/`, February, 15th 2014.

[9] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, January 2005.

[10] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 48–58, 2011.

[11] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. An empirical study of high availability in stream processing systems. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 23:1–23:9, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[14] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23:2351–2365, 2012.

[15] T. Heinze, A. Martin, M. Pasin, R. Barazzutti, C. Fetzer, P. Felber, Z. Jerzak, E. Onica, , and E. Riviere. estreamhub: Elastic scaling of a high-throughput content-based publish/subscribe engine. ICDCS '14, Washington, DC, USA, 2014. IEEE Computer Society.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *In USENIX Annual Technical Conference*.

[17] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. volume 1, pages 574–585. VLDB Endowment, Aug. 2008.

[18] A. Martin, A. Brito, and C. Fetzer. Active replication at (almost) no cost. In *SRDS '11: Proceedings of the 2011 30th IEEE International Symposium on Reliable Distributed Systems*, pages 21 –30, Washington, DC, USA, Oct 2011. IEEE Computer Society.

[19] A. Martin, R. Marinho, A. Brito, and C. Fetzer. Grand challenge: Predicting energy consumption with streammine3g. In *Proceedings of the 8th ACM International Conference on Distributed Event-based Systems*, DEBS '14, New York, NY, USA, 2014. ACM.

[20] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDM Workshops*, pages 170–177, 2010.

[21] D. L. Quoc, A. Martin, and C. Fetzer. Scalable and real-time deep packet inspection. In *Workshop on Distributed Cloud Computing (DCC 2013)*, UCC '13, pages 446–451, Washington, DC, USA, 2013. IEEE Computer Society.

[22] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceeding of the 19th Internationsal Conference on Data Engineering*, pages 25–36, 2003.