Esc: Towards an Elastic Stream Computing Platform for the Cloud

Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Argentinierstr. 8/184-1, A-1040 Vienna, Austria,
{*satzger, hummer, leitner, dustdar*}*@infosys.tuwien.ac.at*

***Abstract*—Today, most tools for processing big data are batch-oriented. However, many scenarios require continuous, online processing of data streams and events. We present ESC, a new stream computing engine. It is designed for computations with real-time demands, such as online data mining. It offers a simple programming model in which programs are specified by directed acyclic graphs (DAGs). The DAG defines the data flow of a program, vertices represent operations applied to the data. The data which are streaming through the graph are expressed as key/value pairs. ESC allows programmers to focus on the problem at hand and deals with distribution and fault tolerance. Furthermore, it is able to adapt to changing computational demands. In the cloud, ESC can dynamically attach and release machines to adjust the computational capacities to the current needs. This is crucial for stream computing since the amount of data fed into the system is not under the platform's control. We substantiate the concepts we propose in this paper with an evaluation based on a high-frequency trading scenario.**

***Keywords*-stream computing; event processing; adaptability**

I. INTRODUCTION

"Cloud Computing" is an ongoing trend in the IT world. It has the potential to transform the way software is used and hardware is designed and purchased. New services can be deployed rapidly without bearing massive initial investments and time delays accompanied by setting up the necessary infrastructure. As pointed out by Armbrust et al. [1], the nearly impossible task of predicting future workload and hardware requirements has become obsolete. Moreover, without a self-maintained infrastructure one can avoid overprovisioning, which would be required to cope with peaks but is a waste of resources in off-peaks.

ESC (pronounced "Escape") is a distributed stream processing platform written in Erlang [2]. ESC employs a similar programming model as MapReduce [3] in that it applies operations to key/value pairs. In contrast to MapReduce, which is tailored to batch-oriented processing of large data sets, ESC is designed to deal with distributed online processing of event streams fed into the system at unpredictable rates. It targets data processing tasks with soft real-time demands such as analysis of sensor network data, online web mining, or algorithmic trading. A directed acyclic graph defines the data flow of a program in ESC; its vertices specify the operations applied to the key/value pairs called events. The usage of DAGs for stream processing is very expressive, much more expressive than an online version of MapReduce. Programmers can focus on the task at hand while ESC contributes concurrency, distribution, and fault tolerance. For large batch-oriented tasks, cloud computing can get results as quickly as the software scales, because using 1,000 servers for one hour causes the same costs as one server for 1,000 hours [1]. This is different for stream processing where we have no control over the arrival rate of events. ESC is supposed to be deployed within a cloud in which it is possible to request and release machines at runtime. In such a setting ESC is able to dynamically adapt to the computational needs, i.e., to automatically scale up and down, to be *elastic* so to say. This is crucial for stream processing in order to cope with peaks and off-peaks.

In this paper we describe the basic architecture, functionality, and programming model of ESC. For evaluation purposes, we use ESC to compute stock correlations in a high-frequency trading setting and present the results. The paper is structured as follows. In Section II we describe the high-frequency trading scenario. Section III explains how stream computing in ESC works, while Section IV focuses on failover and adaptability. We provide insights into the implementation of ESC in Section V. Then, Section VI describes the conducted experiment and the results. Overview of related work is given in Section VII. Finally, Section VIII concludes the paper and points to future work.

II. SCENARIO

In this section we present a motivating scenario from the financial computing domain. High-frequency trading is the trading of financial assets based on computerized strategies with brief holding time and high volume. The correlation among assets plays an important role in algorithmic trading [4]. A simple strategy would be to look at two long-term highly correlated stocks, such as McDonalds and Burger King, and to detect a deterioration in correlation. This could be exploited to make a profit. In simple terms, if the McDonalds stock price increases then you would expect the Burger King stock to gain in the very near future, too. If this is not the case a trading strategy could be to either buy Burger King stocks or sell McDonalds stocks. At the heart

978-0-7695-4460-1/11 $26.00 © 2011 IEEE
DOI 10.1109/CLOUD.2011.27

of such a trading algorithm would be the computation of short-term stock correlation in real-time.

We assume a scenario where, in a preselection step, five promising stocks are selected and the stream computing program is to output the most correlated pair of stocks, according to the Pearson correlation coefficient. Each input consists of the stock symbol as key and a list of transactions' timestamp/price pairs as value.

In order to compute the Pearson correlation, equally spaced data points are needed. However, transactions take place at random time intervals. Therefore, a homogenization step is needed to force equidistantly distributed data points. This preprocessing can be done by simply assigning each data point the closest timestamp's value. We assume in the following that the short-term correlation is based on the ticks of the last ten minutes; we further assume that 50 ticks per second are generated resulting in $30,000$ data points describing one stock. The stock values are provided in the course of a request, i.e., five stock values for each request are identified by the same request identifier. The task is to calculate the most highly correlated stocks for each request.

The scenario introduced above could be implemented in Esc by a DAG as shown in Figure 1. We will denote the vertices of the graph *processing elements* (PEs). Vertices with zero in-degree are called input PEs, zero out-degree vertices are output PEs. The former do not consume events but are pure emitters, bridging external input to the stream processing graph, while the latter represent sinks which make the results available to an external party. Events that are consumed and produced by PEs are composed of a key/value pair. Additionally, events contain an event type and a context field used to join events belonging to the same context. In the high frequency trading scenario the context field is used to group the stocks by request. The input PEs are used as entry point for the stock tick data. The next layer consists of correlation PEs which accept data for two stocks. These PEs perform a homogenization as described above and finally compute the Pearson correlation. The output of the correlation PEs is fed into a PE computing the maximum of all inputs. The final PE outputs the most correlated stocks.

Esc transparently maps and executes the functionality specified by a DAG using machines of the cloud. It adapts to the current requirements imposed by varying input rates and changing computational loads.

III. Stream Computing in Esc

In this section we introduce the main components in the Esc system, and briefly discuss their responsibilities and dependencies. The functionality of PEs, which execute the operators specified in the DAG, are described in more detail. Finally, we show how the operator function of a single PE is specified.

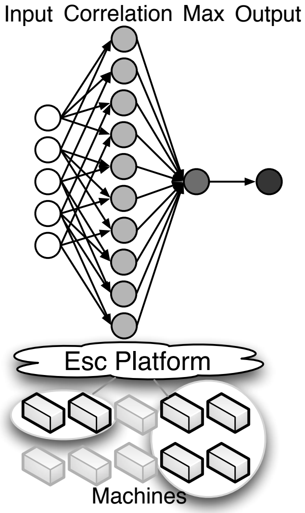

Figure 1. Functionality of Esc: Input PEs create input events, output PEs publish final events, intermediate PEs may consume and produce events; they perform the actual computations. The above shown DAG computes the stocks with maximum correlation based on the tick data of five input stocks. The Esc platform is responsible for mapping PEs to machines in the cloud. It is able to adapt to changing computational requirements, e.g., by adding machines to the resource pool.

A. *Processes in* Esc

In Esc all functionality is implemented by concurrently running lightweight processes. They can be divided into *system*, *machine*, and *processing element* processes, as shown in Figure 2. System processes provide the foundations for running stream computing applications. Systemwide, there is exactly one instance for each such process. Machine processes, on the other hand, need to be running on each node. The processes associated with PEs may run anywhere; a machine can host zero to many PE processes. The *App Info* process takes a user provided DAG and makes it available to all interested processes. It plays a crucial role for initialization and also for failover. The *Pool Manager* is responsible for attaching and releasing machines. It also maintains statistics about all managed nodes, such as their current workload. This information is issued by *Stats Sender* processes and used to map PE processes to suitable machines. *PE Factory* allows for a creation and initialization of PE processes on arbitrary machines of the network in cooperation with a *Supervisor* on the respective machine which creates and monitors them locally. The *Autonomic Manager* component is responsible for high-level adaptation decisions. It collects relevant data and reconfigures the system if necessary. For instance, if all machines of the pool are under heavy load and stream processing performance decreases, it triggers the attachment of a new machine from the cloud if that action is available. The only task of the *Alive Monitor* is

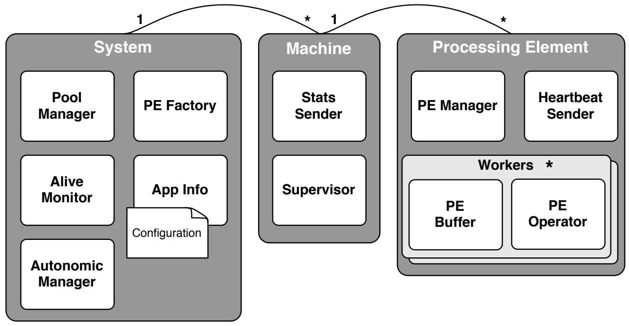

Figure 2. Main processes in ESC: *System* processes provide system-wide functionality, i.e., globally, there is one instance for each system process, running on an arbitrary machine. On each machine, *machine* processes provide basic machine centered functionalities. There may be many *processing element* specific processes running on multiple machines, whereas a processing element comprises a manager process, a monitor process and several workers. Each worker is made up of a PE Buffer/Operator pair.

to monitor input PEs. Input PEs play a special role for the fault tolerance of ESC and are obliged to send heartbeats to the Alive Monitor by a *Heartbeat Sender* process. The Alive Monitor is aware of all input PEs specified by the DAG because of the App Info process and expects to receive heartbeat messages from all of them. PE processes represent the "work horse" processes in ESC which perform the actual stream computing. All other processes can be attributed to service tasks providing configuration, adaptation, and fault tolerance. The *PE Manager* serves as the gateway to the PE it represents, performs load-balancing, and, with the help of the PE Factory, creates and destroys *PE Buffers* and *PE Operators*. There may be many *workers*, i.e., pairs of PE Buffers/Operators, belonging to the same PE. These subcomponents do not need to run on the same physical machine. The PE Factory would typically create new processes on the least loaded machine. The task of the PE Buffer is to cache events and to transmit them to a PE Operator, which performs the actual user-defined computation. PE Operators may emit new events to subsequent PEs, i.e., to their corresponding PE Managers.

B. Processing Elements

PE processes perform the computational task as defined by the DAG. Figure 3 illustrates the functionality of a PE. Its manager process serves as entry point for all incoming events. On arrival of a new event it first checks whether it is responsible for consuming events of that type. PE Managers have an exchangeable balancer function $b : Events \rightarrow Workers$, mapping events to workers. A function $b : e \mapsto 0$ would map all events to the same worker with id 0, whereas a function of $b : e \mapsto hash10(e)$, which hashes e to the range $[1, .., 10]$, would distribute events among ten workers. Each worker's PE Buffer caches events, and a strategy function determines which events are to be joined and when to flush a buffer. One strategy is to join events by their context and flush when a PE Buffer has received an event of a particular context from all incoming neighbor PEs (as defined by the DAG). When a buffer is being flushed, its content, a list of events, is transmitted to the PE Operator, which applies the user-defined operator f to the list of events. Each event has the form $(Type, Context, Key, Value)$. The PE Operator may maintain its state between calls (stateful), as indicated by the database symbol in Figure 3, or not (stateless). Finally, the PE Operator can emit a new event to all successors in the DAG, containing the result of operator f.

C. Programming Model

The programming of a stream computing application consists of two main parts, the specification of the DAG (defining the flow of the stream) and the type of each vertex. Each PE type is assigned a function f which implements the following scheme:

```
f(EventList, State, Args) ->
  {ok, NewState?} |
  {emit, Event, NewState?} |
  {emit_multi, EventList, NewState?}
```

The operator function, called by the ESC system, must have a signature consisting of three parameters. The first is the list of events, the second is a state object, which can be used to implement stateful PEs, and finally a list of arguments. The argument list is empty by default, but arguments can be defined in the configuration. The operator f can either emit no event (`ok`), a single event (`emit`), or multiple events `emit_multi`. When a `NewState` is returned it will be kept and used for the next call of the respective operator. The function for the *Max* operator, which returns the maximum of all inputs, looks as follows:

```
max(EventList, State, Args) ->
  Result = maxVal(EventList),
  {emit, Result}.
```

The function `maxVal(EventList)` picks the event $e = (t, c, k, v)$ with maximum value v in `EventList`. The general idea of ESC is to establish a library of PE types which can be reused. The programmer only needs to draw a DAG with the help of a graphical tool and to either assign predefined types or to create a new type by implementing and specifying an operator.

IV. FAILOVER AND ADAPTABILITY

In this section we elaborate on the initialization and failover functionality of ESC, and how cloud-based adaptability is achieved.

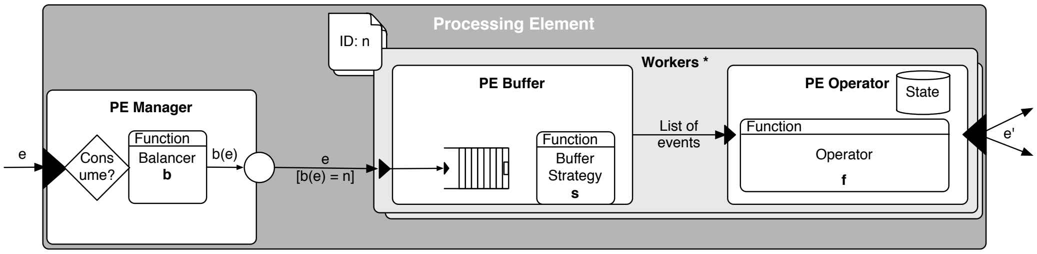

Figure 3. Functionality of PEs. There is exactly one PE Manager responsible for each PE defined in the DAG. It serves as gateway for all incoming events. These are only processed by the PE Manager if the PE has registered for the specific event type. The PE Manager enables load-balancing by defining a balancer function b. PE Buffer caches incoming events and submits the buffered contents to its corresponding PE Operator, as defined by the buffer strategy function s. PE Operator applies the actual operator to the received events and emits new events to all out-neighbors in the DAG.

A. *Initialization and Failover*

For the machine-local detection and restart of crashed processes we use Erlang's supervision trees. A supervision tree is a tree of processes where the parent processes in the tree monitor the child processes and restart them if they fail. In Figure 2, this approach is illustrated in a simplified way. Instead of having one supervisor process per machine there is actually a whole supervision tree on each machine which performs monitoring and restarting of local processes. This local supervision of all processes ensures basic fault tolerance, as long as no machine is going down. All system processes are started as a distributed Erlang application. This means that if the machine running the system process application crashes, the Erlang platform will detect that and restart the application on another machine. Hence, initialization of system processes as well as failover in the case of a machine crash are basically covered by Erlang. For more details about supervision trees as well as distributed Erlang applications and their failover technique we refer to [2]. The Pool Manager requires all attached machines to periodically send heartbeats; if a machine fails to transmit them, it is removed from the pool. When a machine is being attached to the pool, an initial supervision tree is started together with a statistics process, sending heartbeats containing statistics such as the machine's workload.

The above described techniques ensure that all management processes are fault tolerant. In turn, they implement the initialization and fault tolerance of the PE processes, which is described in the following. The basic design principle is that initialization and failover are treated in exactly the same way. The Alive Monitor requires all *input* PEs to periodically send heartbeats. If no heartbeat is received from an input PE for a certain amount of time, it is created. This applies both to the initial phase and to the failure case. The alive monitor detects that certain input PEs are not running and starts them. The initialization phase represents the special case that no single input PE is running, which is however treated equally as if all input PEs were crashed. Whenever a PE emits a new event it sends it to all successors, as defined by the DAG. This includes a short check whether the neighboring PE is actually running - if not, it is created. In more detail, the creation is triggered by the PE which detected the issue and the PE Factory creates the new PE. During that process, the PE Factory communicates with the Pool Manager to figure out the optimal machine, usually the least loaded one. This means that each PE is initially created upon the first attempt to send an event to it. In the initialization phase of a PE, its manager contacts the App Info process to receive information about its configuration, such as the PE's type. At that stage no worker has been created yet. The manager maps each incoming event to a worker ID using its balancer function. It also maintains a hash table mapping from that ID to the respective PE Buffer's address. If a worker ID is not contained in the hash table, the PE Buffer and a corresponding PE Operator are created. Consequently, PE Buffer and PE Operator are created when they are to process the first event that is mapped to that ID. If a failure (e.g., a machine outage) unexpectedly terminates a PE worker, in the worst case the transient data such as the events cached by the PE Buffer and the state of the PE Operator may get lost. The functionality of ESC, however, will be restored if possible, i.e., if a machine for failover is available. We plan to extend ESC with a means to persistently store events and states, which would lead to a highest possible failure tolerance. This feature, however, is likely to influence the overall performance significantly.

B. *Adaptability*

ESC is designed to dynamically adapt to its environment and changing computational needs at runtime. The cloud paradigm that the cost of using 1,000 machines for one hour is the same as using one machine for 1,000 hours implies that a MapReduce job can improve its performance by adding many machines and executing in parallel, while incurring the same monetary cost. This is different for a real-time streaming platform such as ESC. There exists an optimal number of machines the processing is based on, i.e., the minimum pool of machines such that events are processed in a timely manner. In many scenarios events are not fed into the system at equidistant time intervals, but there are peaks and off-peak. For instance, a stream computing platform which is used for analysis of web server logs would typically be confronted with peaks in the daytime and low traffic at night. Therefore *elasticity* is crucial, i.e., capabilities to attach or release resources and to adaptively redistribute processes among the controlled machines.

IBM's vision of its initiative *Autonomic Computing* (AC) [5] is to design computing systems that can manage themselves given high level objectives. The human autonomic nervous system is the inspiration for the term AC, as it is adjusting vital low-level functions such as heart rate and body temperature, allowing our brain to deal with other tasks. The AC initiative introduced the demand on future systems to self-configure, self-heal, self-optimize, and self-protect. Figure 4 illustrates the architecture of ESC's Autonomic Manager, designed according to the MAPE loop [6], proposed by AC as a means for achieving adaptive systems. The monitor phase deals with collection, aggregation, and filtering of information, which is provided by sensors. The preprocessed data are subsequently analyzed. The planner constructs actions needed to achieve goals and objectives. Finally, these actions are carried out by effectors. All four stages of the cycle are based on knowledge the Autonomic Manager has access to.

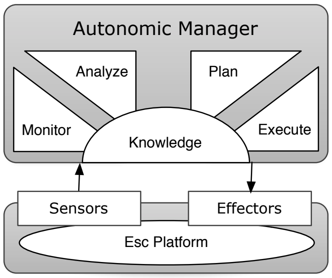

Figure 4. The Autonomic Manager is responsible for adapting ESC to changing conditions - thus realizing its elasticity. It implements a MAPE loop consisting of the steps monitor, analyze, plan, and execute.

In the current version of ESC, the Autonomic Manager has access to two sensors, one providing information about the workload of all currently attached machines, the other delivering data about the queue lengths of worker processes.

The Autonomic Manager has a number of means that have an effect on the performance of ESC and its monetary costs: Machines can be attached to the pool and can also be released subsequently. Also, the balancer function determining the load-balancing within a PE can be replaced during runtime. Replacing a balancer function $b : (t, c, k, v) \mapsto hash2(k)$, which hashes the event's key to one or two, with a function $b' : (t, c, k, v) \mapsto hash5(k)$ hashing the key to the range from one to five would result in the creation of three new pairs of PE Buffer and PE Operator processes, all typically started at the least loaded machine. Much more sophisticated balancing functions are possible, e.g., taking the distribution of incoming data into account. A third way of adaptation is to simply kill worker processes. This results in a restart at the machine with the minimum workload. More drastically but in the same spirit is it possible to kill a whole PE by killing its manager process. In that case all processes related to that PE are killed and will finally be restarted at the optimal location at that time. A powerful further concept for adaptation supported by ESC is *graph rewriting*. Users are given the possibility to define graph rewriting rules along with the DAG. These rules contain information about their effect, i.e., whether they help to increase performance in the case of peaks (expand the DAG) or whether they are suitable to reduce overhead in off-peaks (contract the DAG). A sample rewriting rule is illustrated in Figure 5: it replaces a single sorting PE with a PE splitting the input using a random pivot element, two parallel sort PEs, and a merge PE.

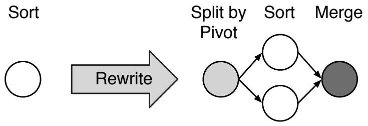

Figure 5. Exemplary graph rewriting rule which divides a simple sorting PE into a sequence of split, sort, and merge.

The monitoring and execution phases of the MAPE cycle are responsible for fetching data provided by the sensors and triggering the effectors, respectively. The analyze component condenses the received information into a higher level representation, according to policies stored in the knowledge base. For instance, the following policy would assign true or false to the logic predicate `overloaded`:

```
overloaded := all nodes have workload > 95%
```

The planning derives actions based on simple "IF condition THEN action" policies, for instance:

```
IF overloaded THEN attach_machine
```

All these instruments can be used to optimally adapt ESC to changing conditions and requirements.

V. IMPLEMENTATION

ESC is written in Erlang, a programming language initially designed at the Ericsson Computer Science Laboratory for building telecoms switching systems. The sequential subset of the language can be characterized as a strict functional programming language which is largely free from side-effects [7]. For concurrency the actor model [8] is employed. Today, Erlang is seen as a general-purpose language and a growing number of projects are implemented in Erlang, such as Amazon SimpleDB [9], Apache CouchDB [10], and Yaws [11]. Also the Facebook chat system is mainly written in Erlang [12].

Erlang was designed for writing concurrent programs, composed of concurrent processes which have no shared memory and communicate by asynchronous message passing. The processes are lightweight and belong to the language, not the operating system [13]. These basic features of Erlang, allowing a large number of concurrent activities, fit well to the needs of our stream computing platform ESC, which in its core consists of concurrently running processing elements. Erlang also makes it easy to distribute the platform over several computers in a network. The soft real-time capabilities of Erlang support the real-time character of stream computing, and its hot code replacement features enable ESC to conceptually run forever evolving over time without restarts. Last but not least, Erlang represents a "battle-proven" solution used in production systems for more than two decades and comes with many powerful libraries.

VI. EVALUATION

For evaluation purposes we have applied ESC to the high-frequency trading scenario introduced in Section II. In a real algorithmic trading application the input PEs would receive tick data from outside by using sockets or similar networking technologies. In the evaluation, however, special input PEs are used which create random tick data events of the form *('stockticks', context, symbol, ticklist)*, e.g., *('stockticks', 55, 'MCD', [(2010-01-31 9:05:10.798, $1.24), . . .])*. The *context* field is an integer incremented by the input PEs for each emitted event; symbol is a random identifier representing a stock and ticklist is a random list of size $30,000$ in which each element consists of a timestamp and a transaction price. A correlation PE takes two events as input *('stockticks', context, symbol1, ticklist1)* and *('stockticks', context, symbol2, ticklist2)*, and outputs *('corr', context, symbol1+symbol2, corr(ticklist1,ticklist2))*. Events are joined based on the context, i.e., the operator is triggered when events from both incoming neighbors have been received which have the same context value. Output events of correlation PEs are labeled *corr*, keep the *context* value, have the key set to a concatenation of *symbol1* and *symbol2*, and a value according to the correlation of *ticklist1* and *ticklist2*. The computation of the correlation is based on the homogenization of both lists whose most costly operation is the sorting of both lists by timestamps. A final value is computed as the Pearson correlation of the homogenized transaction prizes. The max PE reemits the event with the maximum correlation. The balancer functions of each correlation and max PEs is set to $b : (t, c, k, v) \mapsto hash1(k)$, which maps all events to 1, leading to a single worker for each PE; all others PEs use the function $b : (t, c, k, v) \mapsto const$. The evaluation has been executed in a private cloud environment and initially only one node is attached to ESC's pool.

The Autonomic Manager, which is responsible for adaptation, operates according to the following rules (expressed in pseudo code). The analyze phase derives two types of predicates, `mn_overloaded` stating whether all attached machines are heavily loaded and `pe_overloaded(x)` declaring whether a certain PE is overloaded by analyzing the workers' incoming message queues.

```
mn_overloaded :=
 all nodes have workload > 95%

pe_overloaded(x) :=
 one of x's PE Operator processes
  has incoming queue size > 6
```

The planning policies result in attaching a new machine if `mn_overloaded` is true. If no further machine is available the `attach_machine` action has no effect. The second rule defines that if a PE using a hash function for load balancing gets overloaded, then this function is replaced by a function hashing to a tripled range.

```
IF mn_overloaded THEN attach_machine

IF pe_overloaded(x) AND
   x's balancer is hashN(.) THEN
      set x's balancer to hash3N(.)
```

During the evaluation we create requests, i.e., random stock tick data for five stocks fed into ESC, at varying speed. Initially each input PE creates an event every two seconds (on average half a request per second), then the stress level is incremented. The results of the evaluation are illustrated in Figure 6. The x-axis shows the temporal course of the evaluation in seconds. Each cross in the response time section represents a request issued to the stream computing program and the resulting response time. The requests per second part depicts the average number of requests per second and can be seen as the stress level, which is increased over time. The next section shows the number of nodes attached to the pool - initially one node is used. Finally, the figure depicts the number of workers, i.e., the number of PE Buffer/Operator pairs running in the ESC platform.

The response for the first three requests issued to the system takes relatively long. This is due to the nature of ESC to initialize PEs at the time when their functionality is to be used. When the number of requests exceeds one per second, the system becomes overloaded and the response time clearly deteriorates. This situation triggers the attachment of a second node to the pool (see '#Nodes') and the creation of new workers (see '#Workers'). The correlation PEs have to compute the correlation based on long lists and are by far the most heavily loaded processes. Their overload manifests in growing input queues. This causes the *hash1* balancer function to be replaced by a *hash3* balancer functions. In that course, two new workers are

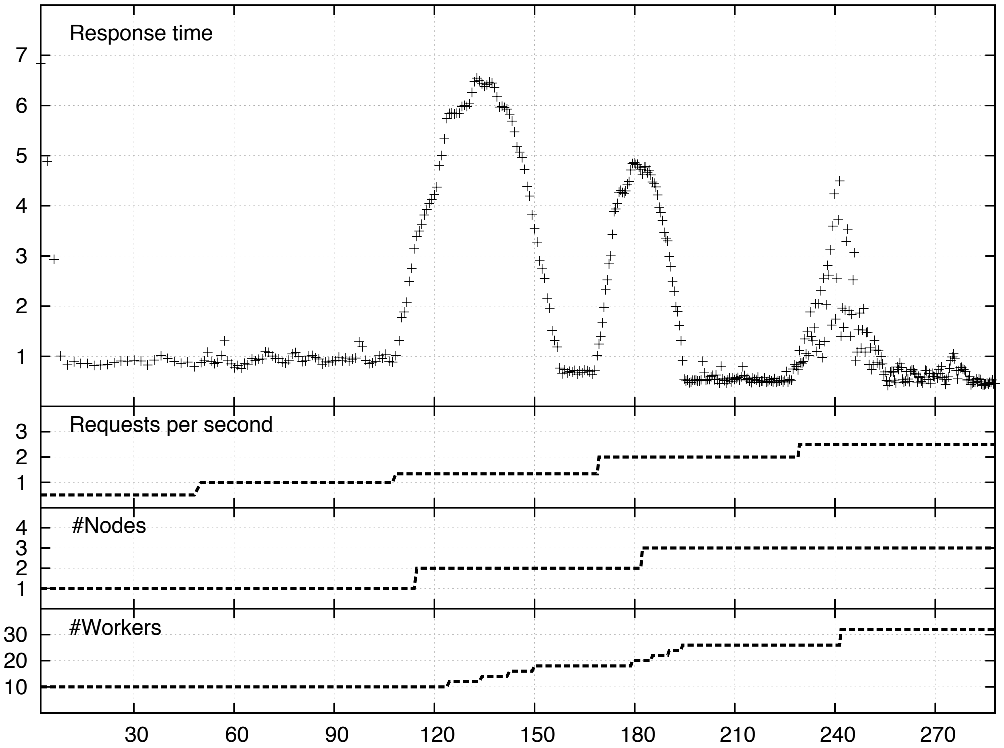

Figure 6. Results of the conducted experiment, a high-frequency trading scenario. Each cross in the section 'Response time' represents the response time in seconds for a single request. A request consists of five stocks each described by $30,000$ tick values. The response contains the two maximally correlated stocks. 'Requests per second' shows the average number of requests per second and can be regarded as the stress level. '#Nodes' depicts the number of nodes attached to ESC's pool. '#Workers' informs about the number of PE Buffer/Operator processes currently running. The system is able to adapt to an increasing stress level by automatically adding nodes and creating new workers.

created on the least loaded machine. These adaptations are controlled by the Autonomic Manager. The increment to two requests per second causes another peak in the response time which is countered by attaching a third node and a further increase in the number of workers. The final peak resulting from the final increase of the stress level can be resolved by increasing the workers. The quite high final jump in '#Workers' illustrates the replacement of a *hash3* balancer by a *hash9* function. All in all, ESC was able to detect overload situations and to dynamically adapt the platform by requesting more machines from the cloud and by increasing the number of concurrently running workers.

VII. Related Work

The Google MapReduce [3] framework is a programming model and an associated implementation for processing and generating large data sets in a batch oriented mode. It is inspired by functional programming, which often uses map and reduce functions. MapReduce's simple key/value based programming model and its scalability are probably the key to its enormous popularity. Dryad [14], a project at Microsoft Research, has similar features to MapReduce, but uses DAGs as a more flexible way of specifying applications. In that respect ESC and Dryad are similar. However, in contrast to ESC, Dryad is explicitly targeted at batch processing and not at online stream processing. In [15], the authors propose modifications of Hadoop, an open-source implementation of MapReduce, that allows data to be pipelined between operators. This extends MapReduce in that it is better suitable to deal with real-time requirements. A similar aim is pursued in [16]. Changes to Hadoop are presented in order to completely remove the barrier between the Map and Reduce stages. This can improve the performance of MapReduce but has a negative influence on the ease of programming. S4 [17] is similar to ESC in that it is designed for stream processing of key/value based events. In contrast to ESC it does not specify programs as DAG and depends on Zookeeper, which provides functionality such as automatic failover. In the current version it lacks adaptability features such as dynamic load balancing. SPC [18] is a platform to support stream-mining applications using a subscription-like model for specifying stream connections. According to [17], SPC is restricted to highly specialized applications. The related area of complex event processing also deals with the analysis of events, but these systems often focus on complex events and queries rather than high throughput and a simple programming model. The STREAM project [19] investigates data management and query processing for long-running queries over streams of data. It deals with building a general-purpose prototype data stream management system and features an own query language. Borealis [20] is a stream processing engine combining the experiences of two projects called Aurora [21], a centralized engine, and Medusa [22], providing distribution.

VIII. Conclusions and Future Work

We have presented our new stream computing platform called Esc. It provides a simple programming model for programmers based on DAGs and hides complexity coming along with aspects such as distribution and fault tolerance. The basic architecture and functionality of Esc have been discussed. We have demonstrated the ability of our engine to dynamically adapt to varying workloads based on a high-frequency trading scenario.

In the future we plan to investigate Esc's capabilities in further scenarios and to compare it to other platforms in more detail. To improve its performance we will examine the possibilities to incorporate graphics processing units (GPUs). In this case Esc would be responsible for the basic execution of a program, but suitable tasks are outsourced to the GPU. Another research direction is to use service level agreements (SLAs) for specifying requirements such as responsiveness. The platform would be responsible for monitoring and enforcing these SLAs. Further links for future work are concurrent execution of multiple programs, heterogeneous environments, quality of data considerations, and more sophisticated adaptation strategies.

Acknowledgement

Financial support by the European Commission (FP7, Grant Agreement no. 257483) is gratefully acknowledged.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, April 2010.

[2] Ericsson AB, "Erlang/OTP System Documentation 5.8.2," http://www.erlang.org/doc/pdf/otp-system-documentation.pdf, Dec 2010.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI 2004*, San Francisco, CA, USA, 2004, pp. 137–150.

[4] M. M. Dacorogna, R. Gençay, U. Müller, R. B. Olsen, and O. V. Picted, *An Introduction to High-Frequency Finance*. Academic Press, May 2001.

[5] P. Horn, "Autonomic Computing: IBM's perspective on the state of information technology." *http://www.research.ibm.com/autonomic/*, 2001.

[6] J. O. Kephart and D. M. Chess, "The vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[7] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, The Royal Institute of Technology, Stockholm, Sweden, December 2003.

[8] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[9] "Amazon SimpleDB," http://aws.amazon.com/simpledb/.

[10] "Apache CouchDB," http://couchdb.apache.org/.

[11] "Yaws," http://yaws.hyber.org/.

[12] E. Letuchy, "Facebook Chat," May 2008. [Online]. Available: ”http://www.facebook.com/note.php?note_id=14218138919”

[13] J. Armstrong, "A history of Erlang," in *HOPL 2007*, New York, NY, USA, 2007.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys 2007*, New York, NY, USA, 2007, pp. 59–72.

[15] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-136, Oct 2009.

[16] A. Verma, N. Zea, B. Cho, I. Gupta, and R. Campbell, "Breaking the MapReduce stage barrier," in *CLUSTER 2010*, Heraklion, Crete, Greece, 2010, pp. 235 –244.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDM Workshops*. IEEE Computer Society, 2010, pp. 170–177.

[18] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: a distributed, scalable platform for data mining," in *DMSSP '06*, New York, NY, USA, 2006, pp. 27–37.

[19] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "STREAM: The Stanford data stream management system," Stanford InfoLab, Technical Report 2004-20, 2004.

[20] 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 *CIDR 2005*, Asilomar, CA, USA, 2005.

[21] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Monitoring streams - a new class of data management applications," in *VLDB 2002*, Hong Kong, China, 2002.

[22] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik, "Scalable distributed stream processing," in *CIDR 2003*, Asilomar, CA, USA, 2003.