



COMPUTING SCIENCE

Scalable and Responsive Event Processing in the Cloud

Visalakshmi Suresh, Paul Ezhilchelvan, Paul Watson

TECHNICAL REPORT SERIES

No. CS-TR-1324

April 2012

Scalable and Responsive Event Processing in the Cloud

V. Suresh, P. Ezhilchelvan and P. Watson

Abstract

Event processing involves continuous evaluation of queries over streams of events. Response-time optimization is traditionally done over a fixed set of nodes using a variety of techniques. The emergence of cloud computing makes it easy to acquire and release of computing nodes as required. Leveraging this facility, we propose a novel, queuing-theory based approach for meeting specified response-time targets against fluctuating event arrival rates by dynamically and adaptively drawing an adequate amount of computing resources from a cloud platform. In the proposed approach, the entire processing engine of a distinct query is used as an atomic unit for optimization and reconfiguration. Several such units hosted on a node are modelled as a multiple class M/G/1 system. These aspects not only eliminate the need for intrusive, low-level performance measurements during run-time, but also offer portability and scalability. The efficacy of the approach is examined through cloud-based event- processing experiments where dynamism and adaptation are shown to be achievable.

Bibliographical details

SURESH, V., EZHILCHELVAN, P., WATSON, P.

Scalable and Responsive Event Processing in the Cloud
[By] V. Suresh, P. Ezhilchelvan, P. Watson

Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1324)

Added entries

NEWCASTLE UNIVERSITY

Computing Science. Technical Report Series. CS-TR-1324

Abstract

Event processing involves continuous evaluation of queries over streams of events. Response-time optimization is traditionally done over a fixed set of nodes using a variety of techniques. The emergence of cloud computing makes it easy to acquire and release of computing nodes as required. Leveraging this facility, we propose a novel, queuing-theory based approach for meeting specified response-time targets against fluctuating event arrival rates by dynamically and adaptively drawing an adequate amount of computing resources from a cloud platform. In the proposed approach, the entire processing engine of a distinct query is used as an atomic unit for optimization and reconfiguration. Several such units hosted on a node are modelled as a multiple class M/G/1 system. These aspects not only eliminate the need for intrusive, low-level performance measurements during run-time, but also offer portability and scalability. The efficacy of the approach is examined through cloud-based event- processing experiments where dynamism and adaptation are shown to be achievable.

About the authors

Lakshmi joined the computing science department as Research Associate in the MESSAGE project in November 2006. She is currently working in TSB funded EVADINE project to build infrastructure for Electric Vehicles Monitoring. She is a member of core technology group to support database infrastructure in SiDE(Social Inclusion of Digital Economy). Her current area of research interests are in building scalable real time event processing system on cloud. She received her Bachelor of Engineering degree in 1996 from the Bharatidasan University, India and the Master of Science(SDIA) with distinction in 2006 from the Newcastle University, United Kingdom. After completing her bachelor's degree she worked for LogicaCMG and Indigo4 Systems United Kingdom in client server technologies. During her post-graduation studies, Lakshmi worked for PrismTech in the Open Splice DDS, a middleware addressing publish-subscribe communications for real-time and embedded systems.

Paul Devadoss Ezhilchelvan received Ph.D. degree in computer science in 1989 from the University of Newcastle upon Tyne, United Kingdom. He received the Bachelor of Engineering degree in 1981 from the University of Madras, India, and the Master of Engineering degree in 1983 from the Indian Institute of Science, Bangalore. He joined the School of Computing Science of the University of Newcastle upon Tyne in 1983 where he is currently a Reader in Distributed Computing.

Paul Watson is Professor of Computer Science and Director of the Digital Institute. He also directs the £12M RCUK-funded Digital Economy Hub on Social Inclusion through the Digital Economy. He graduated in 1983 with a BSc in Computer Engineering from Manchester University, followed by a PhD on parallel graph reduction in 1986. In the 80s, as a Lecturer at Manchester University, he was a designer of the Alvey Flagship and Esprit EDS systems. From 1990-5 he worked for ICL as a system designer of the Goldrush MegaServer parallel database server, which was released as a product in 1994. In August 1995 he moved to Newcastle University, where he has been an investigator on research projects worth over £30M. His research interest is in scalable information management with a current focus on Cloud Computing. Professor Watson is a Chartered Engineer, a Fellow of the British Computer Society, and a member of the UK Computing Research Committee.

Suggested keywords

CLOUD COMPUTING

REAL TIME EVENT PROCESSING

SCALABILITY OF EVENT PROCESSING

Scalable and Responsive Event Processing in the Cloud

Visalakshmi Suresh, Paul Ezhilchelvan, Paul Watson

Event processing involves continuous evaluation of queries over streams of events. Response-time optimization is traditionally done over a *fixed* set of nodes using a variety of techniques. The emergence of cloud computing makes it easy to acquire and release of computing nodes as required. Leveraging this facility, we propose a novel, queuing-theory based approach for meeting specified response-time targets against fluctuating event arrival rates by dynamically and adaptively drawing an adequate amount of computing resources from a cloud platform. In the proposed approach, the entire processing engine of a distinct query is used as an atomic unit for optimization and reconfiguration. Several such units hosted on a node are modeled as a multiple class M/G/1 system. These aspects not only eliminate the need for intrusive, low-level performance measurements during run-time, but also offer portability and scalability. The efficacy of the approach is examined through cloud-based event-processing experiments where dynamism and adaptation are shown to be achievable.

1. INTRODUCTION

Event processing is characterized by the continuous processing of streamed data tuples or events in order to evaluate, in a timely manner, the queries deployed by decision support systems. Event sources can, for example, be pervasive sensors; while the number of sources is normally fixed in an application, the rates at which they generate events can vary widely and often unpredictably, driven purely by the external processes they monitor. Similarly, the number of queries that need to be evaluated over the streams can also vary over time. Thus, an event processing system with real-time performance requirements must meet targeted response times despite being subjected to these two types of varying loads.

A query evaluation can be modeled as a directed acyclic graph wherein nodes are operators and the links are event streams that are either raw or partially processed by the preceding operators. Early commercial systems, such as Aurora [?], used single server solutions and proposed a variety of techniques, such as multi-query optimization, for response-time optimization. Later, distributed solutions [?] handled the optimization problem as a load-balancing issue over a fixed set of nodes: moving query operators to nodes where their resource requirements are best met and thereby achieving the best overall response time. Such solutions however have two drawbacks: they require the placement of low-level probes to measure operator execution rates, queue lengths, etc., making their implementation hard and possibly not portable across heterogeneous

machines; at times, the load has to be ‘shed’ to meet response time targets [?]. In this paper, we present and investigate a novel approach to responsive and scalable event-processing which avoids both these drawbacks; it leverages the advantages offered by, and is best suited for implementation in, cloud computing platforms.

Central to our approach is the way we chose to model event processing activity from a performance analysis perspective. The rationale behind our model can be succinctly explained as below. Incoming tuples in an event processing engine go through a sequence of operators before triggering an output event. The output latency or the *response time* therefore consists of three major components:

- (a) The wait time before encountering the first operator in the sequence,
- (b) The wait time between operators, and
- (c) The sum of operator execution times.

When the arrival rate of tuples increases, wait time (a) is seriously affected. When more engines are hosted by a single computing node, inter-operator delay (b) and operator execution times (c) are impacted due to competition for CPU usage. Based on these observations, we model an event processing engine as a single queue ‘server’ system wherein the server is the composite operator consisting of all operators within that engine. The waiting time in the queue models (a) and the *processing time* by the ‘server’ models the sum of (b) and (c). We use queuing theory to predict queuing time (a). We use off-line calibration to establish inter-operator delay (b) and (c) as the server processing time. Note that (b) and (c) are less affected by variation in arrival rates.

The paper is organized as follows. Next section describes the event processing system that must meet response time targets even when arrival rates can be unpredictably high for long durations. Section 3 presents the overall architecture that our approach warrants and Section 4 highlights the role of the configuration scheduler in finding the optimal number of virtual machines. Sections 5 and 6 present the queuing theory based models and the algorithm for selecting the optimal configuration. Experimental results presented in Section 7 and Section 8 concludes the paper highlighting the validation of the scheduling algorithm for the optimal placement of the event processing system.

2. System Description

The system processes several event streams, each emanating from a distinct source. These streams are denoted as $s_1, s_2, s_3, \dots, s_\sigma$ and complete event streams in the system is defined as $\Sigma = s_1, s_2, \dots, s_\sigma$. The system evaluates q queries, Q_1, Q_2, \dots, Q_q . The state machine that implements the directed acyclic graph, DAG for Q_i is called the event processing network, EPN_i . Evaluating Q_i involves processing one or more event streams and the set of all streams input to EPN_i is denoted as S_i . Note that there is no implication that two EPNs have distinct S , e.g., S_i and S_j may overlap. Also, an input stream to EPN_i can be an output stream from another EPN_j ; if so, $S_i \notin \Sigma$. If all inputs to EPN_i are output streams from other EPNs, then $S_i \cap \Sigma = \{\}$. An EPN is also associated with a performance target T . It is said to be distinct if any one of its three attributes is unique: DAG, S or T . We consider all EPNs to be distinct.

The system itself is made up of n virtual machines, denoted here generically as *nodes*, drawn from a cloud computing platform. The number of nodes used, n , is increased

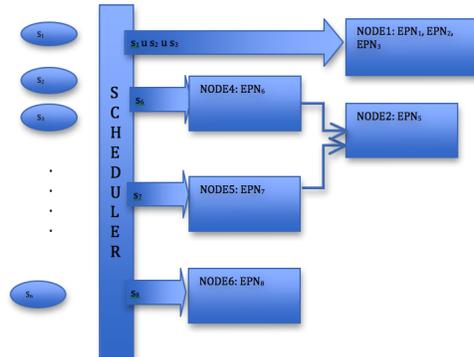


Figure 1. The Event Processing Architecture

(or decreased) when the load increases (or decreases) to an extent that the current configuration over these n nodes is deemed inadequate (or more than strictly necessary, respectively) to meet the performance targets.

A configuration is a mapping from the set of EPNs onto the set of hosts. Figure 1 shows a configuration where, EPN_1 , EPN_2 and EPN_3 are mapped to (e.g., hosted by) node 1, and the rest of the EPNs are mapped to a distinct node. The system has a configuration scheduler, CS for short, which decides the configuration appropriate to the load conditions and performance targets associated with the EPNs. For brevity, we assume that CS is centralized, hosted on a single node. The workings of CS are discussed in section 3.

3. The Architecture

The front end of our system has scheduler, to which all the event sources will be directed to a specific computing node in the cloud according to the policy in the configuration scheduler. The event sources might be included into the system through MOM (message oriented middleware) or asynchronously connected through socket service. When CS announces the EPN-host mapping, each node subscribes to relevant input streams and transmits its relevant output streams, if any, to nodes of EPN which use them as inputs. In Figure 1, nodes 4 and 5 supply their relevant outputs to node 2; all other output streams are archived.

Central to our architecture is the configuration scheduler CS, and an outline of its design is described in section 4. In a nutshell, each EPN takes macro-level measurements of its own performance and reports periodically to CS which constructs a global view and attempts to re-map EPN to host nodes, if response times of some EPN are either above or far below their target levels; in the former case, new nodes may have to be brought in and in the latter some of the existing nodes may be released. Note that re-mapping EPN requires support mechanisms and extracts a cost, both of which are not considered here.

4. Configuration Scheduler

Each node monitors, for every EPN_i deployed within it, the response time RT_i and the arrival rates of each input stream; the maximum RT_i and the sum (AR_i) of the average arrival rates of all its input streams observed over the reporting interval are sent to the CS. For example, node 1 in Figure 1 that hosts EPN_1 , EPN_2 and EPN_3 , will report to CS $\{RT_1, AR_1\}$, $\{RT_2, AR_2\}$ and $\{RT_3, AR_3\}$.

Let T_i denote the average response-time target for EPN_i . If RT_i does not exceed T_i by a specified threshold for all EPN_i , then the current configuration is working well and CS does nothing; otherwise, CS has to decide on a new configuration by dividing the set of all q EPN, $EPN_1, EPN_2, \dots, EPN_q$, into ζ disjoint subsets, Z_1, Z_2, \dots, Z_ζ such that:

1. ζ is the smallest possible, i.e., the number of nodes used in the new configuration is minimum when all EPNs in a given Z_x , $1 \geq x \leq \zeta$, are hosted within a distinct node, and
2. For every EPN in Z_x
 - (a) Each EPN in Z_x meets its target response time, and
 - (b) The total load exerted by all EPNs in Z_x does not exceed the node's capacity.

These two constraints make the new configuration decided by the CS an optimal one. Meeting the first constraint becomes one of *optimal assignment* problem, provided that 2a and 2b can be analytically evaluated (as either true or false) for any given Z_x . Note that the optimal assignment problem is not NP-complete for two reasons: ζ is not fixed and q is finite. We solve it here using a bin-packing algorithm which ‘packs’ the EPNs into the smallest number of nodes (bins), subject to conditions 2a and 2b.

The analytical evaluation of 2a and 2b requires deriving formulae for analytically estimating EPN response times, which in turn makes a simplifying assumption that, a node can host any EPN on its own and also satisfy both 2a and 2b. When this assumption does not hold, Intra-EPN parallelism is necessary, a topic discussed in section 4.1 but not investigated in this paper.

(a) Intra-EPN parallelism

It is possible that we can have a situation wherein EPN_i does not meet its target T_i even though its host node is hosting no other EPN. It can occur, for example, if AR_i is very large. On these occasions, we resort to Intra-EPN parallelism as depicted in Figure 2: EPN_i is hosted on multiple nodes (two nodes in Figure ??) and each input stream in S_i is temporally split and distinct (and temporally disjoint) splits are input to distinct hosts. For example, an input stream s_i can be split as: (t to t+100) tuples as s_i^1 , (t+101 to t+200) tuples as s_i^2 , (t+201 to t+300) tuples as s_i^3 , and so on. The splits s_i^1, s_i^3, s_i^5 are sent to EPN_i^1 (in that order) and the rest to EPN_i^2 , halving the arrival at each destination. The results from EPN_i^1 and EPN_i^2 are to be ‘reduced’ to the final version.

Our approach of Intra-EPN parallelism corresponds to the well-known MapReduce paradigm. In the context of VLDBs, it is known as intra-operator (or partitioned) parallelism [?]. The existing solutions for intra-operator parallelism use multi-query optimization by scheduling the incoming workload in a fixed number of nodes. Operators are shared by several queries based on a dynamic data scheme to maximize resource utilization.

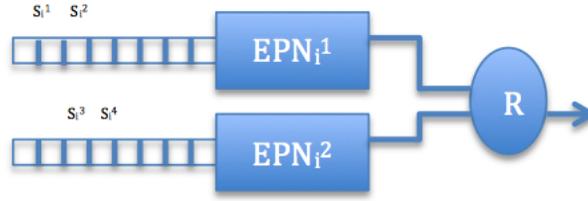


Figure 2. Intra-EPN parallelism

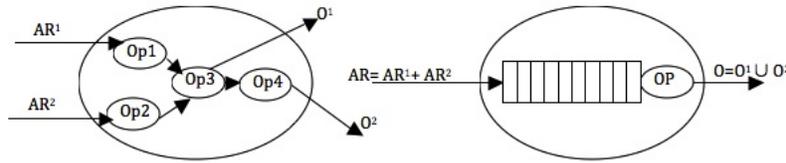


Figure 3. Internals and model of an EPN

5. Analytical Estimation of EPN Response Times and CPU usage

With no loss of generality, let us consider a single Z_x consisting of k EPNs: $EPN_1, EPN_2, \dots, EPN_k$. Each EPN_i where $1 \leq i \leq k$, is modeled as a single queue server system as explained below.

Figure ?? presents a typical DAG structure of an EPN with two input streams with arrival rates AR^1 and AR^2 . After these streams are acted on by distinct operators (Op1 and Op2), they are joined at Op3 which produces an output stream O^1 to the environment and an input stream to Op4 which then generates O^2 . Note that each Op would have its own buffer to store the incoming events directed at it and these buffers are not shown in Fig ??.

Irrespective of its internal structure, an EPN is modeled as a single server system that receives a single input stream with an arrival rate $AR = AR^1 + AR^2$, and generates a single output stream $O = O^1 \cup O^2$; the incoming event tuples are queued and the tuple at the head of the queue is processed by a super operator OP composed of Op1, Op2, Op3, Op4. The tuples that get past the head of the queue, in our model, are ‘processed’ by OP as per the logic of EPN being modeled to generate O.

For each EPN_i , $1 \leq i \leq k$, we define or recall the following metrics, some of them measured dynamically and some others established through calibration.

Event Arrivals denotes the arrivals of events of streams in S_i and are taken to be Poisson at the rate of AR_i which is supplied to CS at the end of each reporting interval.

Processing Time denotes the total processing time a tuple or a window of tuples needs to undergo to produce an output $O^j \in O$ after a tuple in the window has just gone past the head of the queue as in figure ?. Note that it does not include the time that a tuple spends between its arrival and reaching the head of the queue *viz.* the *queuing delays*. Moreover, it depends on the nature of DAG_i that EPN_i implements and a particular path that a tuple takes within DAG_i for it to be processed and an appropriate output to be generated. Given

the non-deterministic nature of tuple-flows, processing time is appropriately modeled as a random variable of *arbitrary* (or some unknown) distribution. When EPN_i generates several outputs, the one that takes the maximum processing time will be of our interest. Let b_i denote the *average processing time* for the most processing intensive output and $M_{2,i}$ the second moment of the processing times.

Processing Load imposed by EPN_i is ρ_i . Specifically, ρ_i is the fraction of the time the node allocated the processing engine to EPN_i . Thus, $\rho_i = AR_i \times b_i$ and this relation is used to establish $M_{2,i}$ and b_i are through *calibration* as described below.

EPN_i is hosted on its own on a node and subject to ‘small’ arrival rates AR_i (to eliminate or at least minimize queuing delays); the cpu usage noted for each rate is observed and the resulting processing times are worked out from these observations. From these times, the most processing intensive output is identified, and $M_{2,i}$ and b_i are established. EPNs typically process input events in groups or windows of, say, w tuples; if so, the arrival rate during calibration should not exceed w events per second.

When a single node hosts k EPNs of Z_x , we model it similar to the way the collection of operators of a single EPN was modeled, the k EPNs of Z_x , are replaced by a *composite* OP of all k EPNs and the input events of various EPNs are placed in a single FIFO queue; when an event or a window of appropriate events gets past the head of the queue, the (virtual) processor of the node processes it by executing the OP of the appropriate EPN_i . The model thus becomes the M/G/1 multiple class queuing system [6] (which necessitates using the second moments of processing times). So, for a node hosting k EPNs of Z_x ,

1. the overall arrival rate, $AR = AR_1 + AR_2 + \dots + AR_k$
2. the overall load on the node $\rho = \rho_1 + \rho_2 + \dots + \rho_k = b_1 \times AR_1 + b_2 \times AR_2 + \dots + b_k \times AR_k$
3. the overall second moment, $M_2 = \frac{1}{AR} (M_{2,1} \times AR_1 + M_{2,2} \times AR_2 + \dots + M_{2,k} \times AR_k)$
4. The average response time W_i for EPN_i to service an input event = $(AR \times M_2) / (2(1 - \rho)) + b_i$

The first term in the expression of W_i (Equation 4) denotes the average queuing delay for an event or a window.

If, for all EPN_i , $1 \leq i \leq k$ in Z_x ,

- *low water-mark* $\leq \frac{(W_i - RT_i)}{RT_i} \leq$ *high water-mark* **(5)**, and
- $\rho = \rho_1 + \rho_2 + \dots + \rho_k$ is less than 1 **(6)**

then deploying these k EPNs of Z_x in a single node is a viable option; otherwise, not. Using (5) and (6), 2(a) and 2(b) of Section 4 can be easily evaluated for any given subset Z_x of EPNs, provided the following hold:

- The node is a single CPU or a single core machine, and
- The CPU of the node being considered for the possible hosting of EPNs of Z_x is equally powerful as the CPU of the node used for the calibration of EPNs.

If the former is more powerful, the estimates of response times would be larger (than the actuals) and hence the evaluation of 2(a) and 2(b) would be pessimistic which would result in using more VMs than strictly necessary. On the other hand, if the calibration is carried out using nodes with a powerful CPU, response-time targets may be missed frequently.

6. Optimal Configuration Selection

Recall that each EPN_i in our system is represented within CS by five parameters: the observed response time RT_i (as periodically reported to CS), the target response time T_i (given), the observed total arrival rate AR_i (reported to CS), the processing time b_i (calibrated) and an estimated response time W_i . When CS observes that one or more EPNs have their $\frac{(RT_i - T_i)}{T_i}$ exceeding the high watermark or falling below the low watermark, it would seek a different, optimal configuration. This selection process itself does not require halting of any EPNs and can proceed in parallel; however, once a different optimal configuration is found, implementing the latter requires moving some EPNs to different nodes and re-directing event streams to appropriate nodes; the latter may involve duplicating a stream in the extreme. In what follows, we describe an algorithm for determining an optimal configuration and ignore the cost of implementing this configuration which would be a topic of our future research. A higher level decision system may find the reconfiguration cost excessive in relation to the performance benefits and decide to stick with the current configuration for a few more reporting intervals.

The reconfiguration algorithm associates each working node N_i with a list E_i of all EPNs hosted in N_i . Nodes that host one or more EPNs with $\frac{(RT - T)}{T}$ exceeding the high watermark is marked as donor nodes and entered in the donor list; the rest of the working nodes are marked as acceptor nodes and entered into the acceptor list. In the new configuration, an acceptor may take in more processing load and a donor must rid itself of some or all of its under-performing EPNs which it is currently hosting. (An EPN under-performs when its $\frac{(RT - T)}{T}$ exceeds the high watermark and is the most under-performing one if its $\frac{(RT - T)}{T}$ is the largest among the EPNs co-hosted with it.) The algorithm has two phases.

The purpose of the first phase is to empty the donor list and is skipped if the list is empty to start with. The donor list is always kept arranged in the non-increasing order of the total cpu load (ρ) that the EPNs impose on the donor nodes (see (2) in Section 5). The acceptor list, on the other hand, is always kept arranged in the non-decreasing order of the nodes' total cpu load. Further, the list E of EPNs for each donor is kept arranged in the non-increasing order of their $\frac{(RT - T)}{T}$. Thus, the first node in the donor list, say node D, is always the most heavily loaded, the first node in the acceptor list is always the least loaded, and the first EPN in the D's E list, E_D , is the most under-performing EPN in node D. This EPN ought to be moved out of D and is denoted as EPN_D .

Starting with the first node in the acceptor list, each acceptor node A is tried by checking if $E_A \cup EPN_D$ satisfy the constraints 2(a) and 2(b) by using (5) and (6). The first acceptor found to meet these constraints becomes the new host for EPN_D . If no node in the acceptor list is found to satisfy these constraints, a new node is to be hired from the cloud to host EPN_D .

Let the new host for EPN_D be denoted as the candidate node N_C . In the new configuration, E_D is set to $E_D - EPN_D$ and E_C to $E_C \cup EPN_D$. (Note that if N_C is to be hired fresh, its E_C should be initialized to $\{\}$). N_C is entered into the acceptor list, if it is to be a newly hired node. Using (5) and (6), the donor status of node D is assessed and D is either retained in the donor list or moved to the acceptor list accordingly.

Nodes D and N_C giving up and gaining EPN_D respectively call for re-arranging the lists and identifying new D and EPN_D . This process of finding a new host for EPN_D is repeated so long as an EPN_D exists, i.e., until the donor list remains non empty.

The purpose of the second phase is to attempt to reduce the number of acceptor nodes, when the latter is more than one. In this phase, a deuterio-acceptor list is created as a copy of the original without the first node (i.e., the least loaded one). The first node is treated as a pseudo-donor node D whose EPNs are all considered as under-performing ones. The algorithm of the first phase is repeated to move the EPN_D of the D node to one of the acceptor nodes in the deuterio-acceptor list. However, if EPN_D cannot be moved to any of the nodes in the deuterio-acceptor list, then a new node is not hired but the attempt is considered to have failed. On the other hand, if all EPNs of D can be moved using only the nodes in the deuterio-acceptor list, then the latter becomes the new acceptor list and a new deuterio-acceptor list is created. The process is repeated until a failure is encountered or the deuterio-acceptor list is empty. The acceptor list and the E lists of the nodes in it provide the new optimal configuration. Implementing the new configuration would involve hiring new nodes (if at all), moving some EPNs and redirecting input streams.

7. Validation

EPNs used for validating the configuration selection algorithm are activity recognition engines that process events emanating from sensors embedded within utilities in Newcastle Ambient Kitchen project [3]. There are multiple utility classes, such as table spoon, tea spoon, fork, knife for eating, knife for chopping, floor boards, cupboard doors so on; there can be several objects of a given utility class. Thus, each of the input streams, $s_1, s_2, s_3, \dots, s_\sigma$ (see Section 2), emanates from a distinct object in the ambient kitchen and σ can be around 600.

The Event processing networks (EPNs) are deployed in an Amazon elastic compute cloud which provides the flexibility to hire various instance types to meet the computing need. Each instance provides a predictable amount of computing power which is charged on an hourly basis. For the evaluation of the scalability and responsiveness, a machine image (AMI) was created and stored in Amazon s3. The Amazon machine image was built with the instances of the event processing networks and the configuration files. The Amazon machine image was built on the EC2 small instance with 1.7GB memory, 1EC2 compute unit, 160GB instance storage, 32 bit platform with Ubuntu operating system. Experiments are run in two stages: (i) calibration of EPNs and the verification of the calibrated value, (ii) and selecting an optimal configuration from a variety of initial configurations deliberately kept as sub-optimal.

Each event producer generates the streams of data denoting an event arriving and entering a data window for aggregation. This experiment uses a window of computation based on the order of arrival of the incoming event streams. As an experiment objective, it is instructed to keep only latest 64 events of a stream in one computational window. The window size of 64 has been determined as optimal by the earlier, activity recognition experiments [5]. The system enters all the arriving events based on the sequence of arrival into a window. When the window is full, the oldest 32 events are pushed out of the window. The system keeps record of all incoming events as new and all events leaving the window as old.

The figure ?? illustrates how the window content changes as the events arrive and the aggregation mechanism computes the features. A given EPN_i receives a distinct subset S_i of these streams as its input; for each input stream, EPN_i typically builds windows of 64 events, maps each window into an attribute tuple and compares the attribute tuples

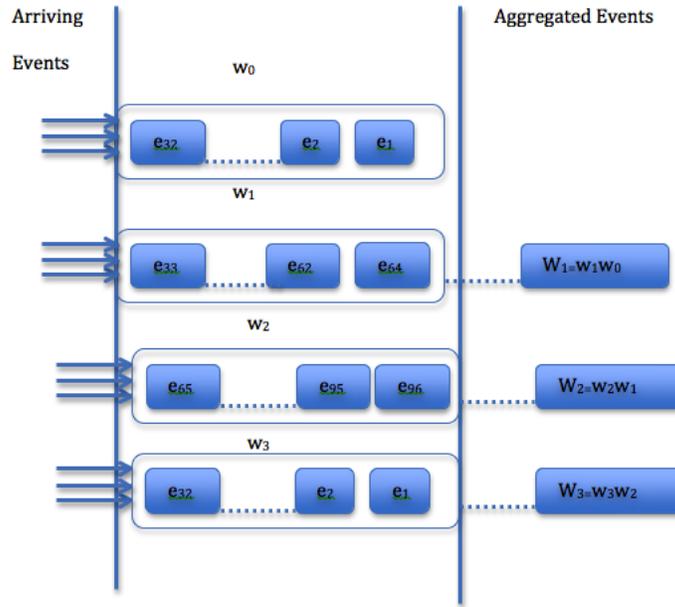


Figure 4. Sliding Window Based Event Processing

with a utility class specific template to identify the activity of the event source. Arrival rate peaks to a maximum when the source is under maximum use and drops to zero when at rest. Thus, EPNs can face a fluctuating event arrival pattern (assumed to be Poisson with rate AR_i - see Section 4).

Given the nature of event processing, we assume that the processing times within any EPN_i are exponentially distributed with mean b_i . This assumption simplifies the second moment $M_{2,i}$ to $2(b_i)^2$, leaving only b_i to be established by calibrating EPN_i which is done as follows. For a range of small AR_i values, EPN_i was run on a single node, and the load ρ_i imposed in each case was observed. The average of the values computed using ρ_i/AR_i , is taken as b_i .

(a) Single EPN on a Single Node

To start with, we hosted a single EPN on a single node for calibration using a low event arrival rate below 40 events per second; the average processing time of the EPN was established as b (see Section 5). The accuracy of the calibrated b was assessed by running the experiments as described below.

The response time RT was measured by varying the arrival rate AR in a larger range (up to 1000 events per second). For each AR , we analytically estimated the response time W using equations (1) - (4) that are modified for the case wherein a node hosts only one EPN - the calibrated one. The modification simplifies (4) to: $W = b [\rho / (1 - \rho) + 1]$. (Note: this simplification does not hold when several EPNs are hosted in a single node.)

Figure ?? plots the measured and estimated responses time for various arrival rates. Ideally, $W = RT$ and, in practice, it is often not. When AR gets larger, $W > RT$. Recall that in the optimal configuration selection, W would be used to check whether the target

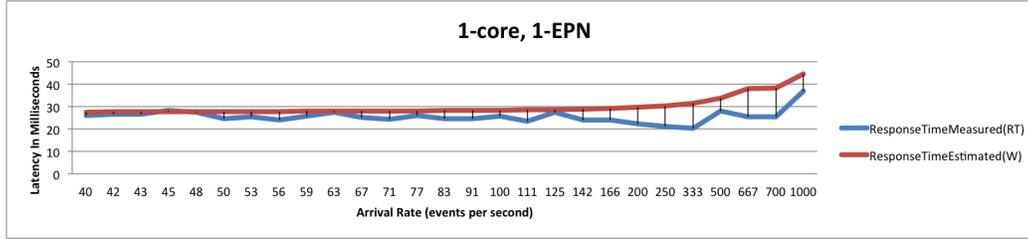


Figure 5. Response Times Measured and Estimated

response time T would be met in a given situation. This means that when $W > RT$, actual response times in the new configuration are unlikely to exceed T and, equally, the number of nodes used may be strictly more than needed.

(b) Configuration Selection Case-Studies

The efficacy of the algorithm for selecting optimal configuration is examined by considering three scenarios. Each involved generating a sub-optimal configuration by loading a selected set of EPNs and then checking if the algorithm selects an appropriate alternative configuration. The three scenarios considered involve

1. Shifting some EPNs among the existing computing nodes so that target response times can be met.
2. Scaling up the number of nodes used so that all EPNs can meet their target response times.
3. Scaling down of the number of nodes used without any EPN missing its target response time.

We fixed the low water-mark and high water-mark to be -100% and 20% respectively in our experiments. That is, an EPN with response time RT and target T would be considered to be performing well, if $-1 \leq \frac{RT-T}{T} \leq 0.2$.

(b.1) Scenario 1: Shifting EPN

The initial configuration has two nodes: node 1 has 3 EPNs with arrival rates of $AR_1 = 500$, $AR_2 = 1000$, $AR_3 = 200$, and node 2 has just one EPN with arrival rate of $AR_1 = 333$. Node 1 has one EPN whose RT exceeds the high watermark deviation and is shown in red in Table ???. The configuration selected by the algorithm consists of the under-performing EPN having been moved to node 2; the response times of all EPNs meet their targets in the new configuration.

(b.2) Scaling up the number of nodes

In this scenario, the selection algorithm suggests the use of three nodes instead of two computing nodes as in the original configuration (see Table ??, scenario 2). The under-performing EPN (shown in red) with $AR = 200$ is shifted to the new (third) node to be hosted on its own.

Arrival Rate (AR) in events/second	Response Time (RT) in ms	Target (T) in ms	(RT-T)/T in %
SCENARIO 1(SHIFT EPN): BEFORE RECONFIGURATION			
500	91.4	100	-8
1000	89.27	120	-34
200	188.38	90	52
333	20.85	30	-47
SCENARIO 1(SHIFT EPN): AFTER RECONFIGURATION			
500	122	100	18
1000	102	120	-17
333	30	30	0
200	67	90	-33
SCENARIO 2 (SCALE UP): BEFORE RECONFIGURATION			
500	91.4	100	-8
1000	89.27	120	-34
200	188.38	90	52
500	122	100	18
1000	102	120	-17
SCENARIO 2 (SCALE UP): AFTER RECONFIGURATION			
500	122	100	18
1000	102	120	-17
500	122	100	18
1000	102	120	-17
200	22.6	90	-304
SCENARIO 3 (SCALE DOWN): BEFORE RECONFIGURATION			
90.9	61.03	80	-31
250	36.55	90	-146
111	23.49	30	-27
SCENARIO 3 (SCALE DOWN): AFTER RECONFIGURATION			
90.9	44.09	80	-89
250	69.67	90	-29
111	27	30	-11

Figure 6. Configuration Algorithm Results

After being shifted, the EPN with $AR = 200$ falls below the low water-mark threshold considerably (shown in yellow). Recall that the selection algorithm, in its second phase, attempts to reduce the number of acceptor nodes, after having reduced the number of donors to zero. In this scenario, the second phase did not succeed. The rationale is obvious if we look at the original configuration of this scenario. The EPN with $AR = 200$ cannot obviously be retained in node 1 (where it is already under-performing) and node 2 already has an EPN with $AR = 500$ which is close to exceeding the high water-mark threshold of 20%; this means that EPN with $AR = 200$ cannot be safely hosted in node 2 as well and hiring node 3 is inevitable, even though the capacity of node 3 is not fully utilised.

(b.3) *Scaling down of hired computing nodes*

In this final scenario, only three EPNs are hosted in 2 nodes and each EPN is deliberately subjected to low arrival rates. The EPN with $AR = 250$ falls below the low water-mark threshold and the algorithm suggests that all three EPNs be placed in node 1 and node 2 be released. In the new configuration no EPN exceeds the high water-mark threshold.

8. Conclusions

We have outlined and experimentally evaluated an approach for deploying an event processing system on Cloud platforms in a scalable and responsive manner. Experiments confirm that optimal configurations can be dynamically identified without any need for non-intrusive, low-level measurements; periodic reporting of arrival rates and response times by EPNs is the only overhead imposed. This small but inevitable running overhead make our approach a highly scalable one in managing a large number of EPNs with target response times.

Ishii and Suzumura [7] recently addressed the issue of hiring VMs from a cloud infrastructure on the need basis. Unlike us, they do not estimate the likely response times for the prevailing arrival rates but use only the arrival rate estimates as a parameter for deciding on the nodes needed; rate estimates are predicted based on the prevailing rates. Their optimization algorithm also takes the cost of hiring extra nodes as another parameter.

The work presented here has two limitations, in addition to not considering the cost of hiring extra nodes. The major one, that is currently being addressed, is that the nodes considered here ought to be only 1-core; the cloud infrastructures offer 4-core VMs and our models are being extended for multi-CPU nodes. The second limitation is that the EPNs used for validation are simple in structure, consisting only of a linear chain of just three operators, and are also identical. This came about due to the choice of experiments for evaluation was driven by the real world application of the ambient kitchen [3] project being carried out at Newcastle. That said, we believe that not considering non-identical EPNs with nonlinear DAG of operators for validation is in itself not a serious drawback as the M/G/1 model is long established and well tested in many other application contexts.

Currently, we are implementing the architecture to manage a distributed, large-scale system of 12 ambient kitchens in multiple locations each with 600 devices and hence requiring 600 EPNs. This would stretch our approach for extreme scalability and the

configuration scheduler itself will have to be distributed. This and other challenging issues are to be addressed in future and the work presented here forms the foundation for the task ahead.

References

- M. Balazinska, H. Balakrishnan, and M. Stonebraker, Load management and high availability in the medusa distributed stream processing system *In Proceedings of 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04
- Manish Mehta, David J DeWitt, Managing Intra-operator Parallelism in Parallel Database Systems, Proceedings of 21st VLDB conference, Zurich, Switzerland, 1995.
- C. Pham, T. Plotz, and P. Olivier, A dynamic time warping approach to real-time activity recognition for food preparation, *In Proceedings of the First international joint conference on Ambient Intelligence*, Aml'10, pages 21-30, 2010.
- Zhou, Y. Ooi, B. C. Tan, K.-L. Wu, J., Efficient Dynamic Operator Placement in a locally Distributed Continuous Query Systems, *published in springer-verlag, lecture notes in computer science, 2006*, numb 4275, pages 54-71.
- Daniel J. Abadi, D Carney, U Cetintemel, M Cherniack, C Convey, S Lee, M Stonebraker, N Tatbul, S Zdonik, "Aurora: a new model and architecture for data stream management", *Springer-verlag*, 2003
- Erol Gelenbe and Isi Mitrani, Analysis and Synthesis of Computer Systems (Second Edition), Volume 4, Advances in Computer Science and Engineering, Imperial College Press, 2010, ISBN: 9 78 1848 168959
- Atsushi Ishii and Toyotaro Suzumura. Elastic Stream Computing With Clouds. IEEE 4th International Conference on Cloud Computing, 2011, pp. 195-202.