Distributed Real-time Event Analysis

Julian James Stephen [α] Daniel Gmach [β] Rob Block [γ] Adit Madan [β] Alvin AuYoung [β]
[α] Purdue University, West Lafayette [β] HP Labs, Palo Alto [γ] HP Software, Sunnyvale

***Abstract*—Security Information and Event Management (SIEM) systems perform complex event processing over a large number of event streams at high rate. As event streams increase in volume and event processing becomes more complex, traditional approaches such as scaling up to more powerful systems quickly become ineffective. This paper describes the design and implementation of DRES, a distributed, rule-based event evaluation system that can easily scale to process a large volume of non-trivial events. DRES intelligently forwards events across a cluster of nodes to evaluate complex correlation and aggregation rules. This approach enables DRES to work with any rules engine implementation. Our evaluation shows DRES scales linearly to more than 16 nodes. At this size it successfully processed more than half a million events per second.**

Keywords—Distributed event analysis, enterprise security;

I. INTRODUCTION

Today enterprises need to cope with a vast amount of diverse data. Several trends are contributing to this data tsunami. First, is the impending reality of the Internet of Things—Gartner predicts that there will be 26 billion smart devices connected to a network by 2020 [1]. Each of these devices will be able to produce and send data of various types and on various timescales. Second, Web sites (e.g., social networks) track increasingly fine-grained user behavior and capture every user action in order to increase the ability to personalize or monetize user experience. Third, enterprises monitor their IT infrastructure, associated software solutions, and services in real-time. Moreover, each component (asset) within an IT infrastructure may generate several different event streams.

The proliferation of autonomous data monitoring, data-processing and data-storage software has given enterprises an unprecedented level of visibility into these data. For large enterprises with many such event streams, there is a significant technical challenge to maintain the systems that monitor, manage and secure their entire pool of IT assets. The primary challenge facing these enterprises is to ingest and analyze an increasingly large volume and variety of data in real-time, and to do this on such a granularity to create immediate and actionable insights.

For example, generating dynamic firewall rules based on a single malicious stream of network traffic—out of possible tens of thousands of benign streams—is a critical and time-sensitive task required to protect the security of the enterprise's IT infrastructure. In a production scenario, the detection rules and corresponding actions are often much more complex, requiring complex rules written by operators with significant domain expertise. Common batch oriented data-processing frameworks used for scalable data ingestion and analysis, such as Hadoop [2] are unable to provide results in real-time. Newer real-time data-streaming frameworks such as Apache Storm [3] or Spark Streaming [4] only support generic compute models, and therefore require significant event processing logic to be embedded in code that must then be reviewed by an IT specialist. In practice, this makes it very difficult to incorporate the knowledge and intuition of human domain experts. As a result, most enterprises rely on more sophisticated systems for Security Information and Event Management (SIEM). These systems process and analyze data streams in real-time to detect interesting events or patterns. They are explicitly designed to allow event processing logic to be specified as rules in a format that is intuitive and easily understood, reviewed and edited by domain experts rather than IT experts. This enables rapid development and easy maintenance of the system.

Unfortunately, current SIEM systems are not scalable. The nature of their design requires a centralized software architecture. To understand why, we provide a brief sketch of a typical SIEM system.

A SIEM system performs automated event processing on data streams and generates customizable output for a system infrastructure or network administrator. In the IT management scenario, the SIEM system typically connects to various data sources (e.g., system loggers, HTTP loggers) and allows an administrator to specify patterns in the form of predicates or rules to detect noteworthy events within the data stream. A basic rule typically consists of three parts: (i) event type (e.g., a HTTP log entry), (ii) conditions that need to be met (i.e., predicates on event attributes, such as "ip_address = 10.10.10.10"), and (iii) actions that will be triggered if an incoming event matches a particular type and predicates are satisfied. More sophisticated rules might specify conditions that invoke actions only for a minimum number of relevant events within a sliding time window (this is referred to as "time aggregation"). Similarly, conditions can range in complexity, from simple string matching to a lookup within an updatable internal data structure. Another complex feature of such systems are rules that use "event joins". These join rules correlate events of different event types, such as correlating HTTP log events with DNS log events. For effective event analysis, SIEM systems should be able to not just process the current stream of events but also be stateful. Statefulness in this context means to store relevant state in internal data structures for future reference. This is important because the significance of future events may change depending on past events. This can be achieved with lookups in the internal data structures.

However, current SIEM systems that support stateful event correlation and aggregation are centralized software systems. A centralized system is inherently a serialization point for incoming data, thereby significantly limiting the rate that events that can be ingested.

978-1-4673-6971-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ICAC.2015.12

We present the design and implementation of a distributed, rule-based event evaluation system, called DRES (Distributed Rule Evaluation System). This system provides similar functionality to a centralized SIEM system, while using an architecture that can scale-out across a cluster of commodity servers. The distributed architecture permits a more cost-effective approach to scaling system throughput. In practice, we observe that when using a more traditional scale-up approach to increase throughput, marginal cost of doubling the throughput increases significantly for higher throughput values. This is because servers with twice the CPU and memory capacity typically cost significantly more than twice the amount in dollars. If we are able to provide higher throughput using a cluster of cheaper, commodity servers, we can scale the system throughput at lower cost.

The key contributions from this paper are:

- The design and implementation of a distributed, rule-based event evaluation system that can scale with the volume of the input event streams.
- An evaluation of our system using production test workloads from a real enterprise security management product. We present detailed performance results that show that our prototype scales linearly with the number of available servers.
- A discussion of the trade-offs and design decisions of providing scalable event-processing in the context of SIEM systems.

The remainder of this paper is organized as follows. Section II introduces the event flow of typical SIEM systems using a simple example rule set. Section III describes the architecture and design of our DRES prototype and Section IV presents empirical evaluation results. Section V explores related work and Section VI concludes with final remarks and future work.

II. Rule-Based Event Evaluation

This section describes how a centralized rule-based event evaluation system works and defines the terms used in the rest of the paper.

An *event* represents any happening of interest monitored by the system. Our event is a collection of key-value pairs. The key is also referred to as *event field* and the value as *event value*. A set of events with *n* key fields can be represented as $E<f_1, f_2 \ldots f_n>$ and a specific event can be represented as $e<v_1, v_2 \ldots v_n>$. In this notation, f_i represents the name of the event field and v_i represents the value of an event *e* for the corresponding field f_i. Events are reported to the SIEM system by *connectors*, which are located at the source of the event. For most SIEM systems there exist connectors to a wide variety (>350) of commercial applications, such like DNS servers, firewalls, anti-virus/anti-spam software, etc. Connectors convert events into a common event format that is then forwarded to the SIEM system. The SIEM system ingests these events and reacts accordingly.

A set of *rules* defines what happens when an event is ingested by the SIEM system. For each event, all rules are evaluated. A *simple rule* comprises of a condition and *action.* Action refer to any activity that is triggered when the rule

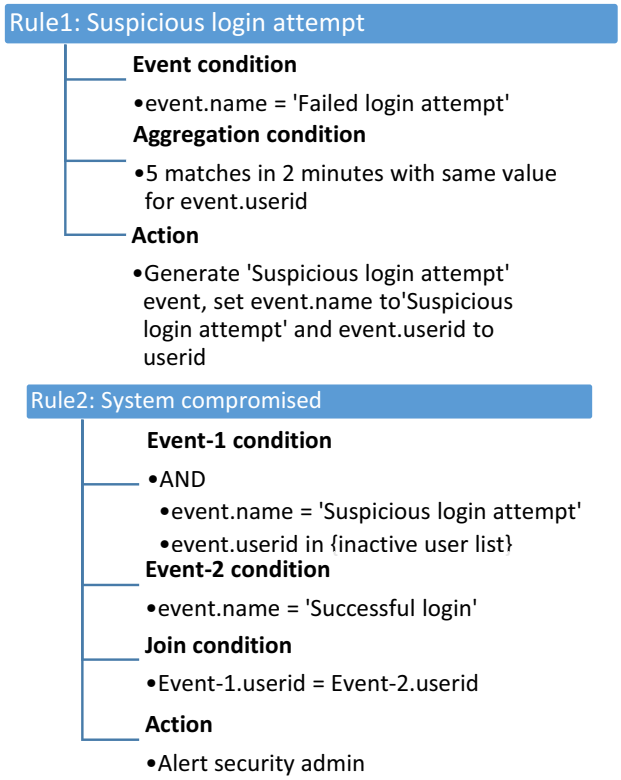

Figure 1: Example rules

condition is evaluated to true. We refer to such rules as simple rules because they are fully evaluated for each event and their outcome is not dependent on events other than the one being processed. A rule condition can consist of sub-conditions that are combined with logical operators such as AND or OR. In addition to simple rules there are more complicated rules that require multiple events to trigger actions. First, there are *join rules*. Join rules correlate two or more events. For example, a join rule may correlate a "*multiple failed logins attempts*" event with one "*successful login attempt*" event to determine suspicious logins. Join rules have the structure $<E_1, E_2, \ldots, E_n, J>$, where every E_i is a condition representing a specific kind of event and J represents the join condition that correlates events of different kinds. The join condition J can also be composed of sub-conditions. Second, there are *aggregation rules*. Aggregation rules identify groups of events with identical and/or unique field values over a specified time frame. Rules can also have join and aggregation components in them at the same time.

Figure 1 shows two example rules. The first rule, '*Rule1*: *Suspicious login attempt'*, checks if the value of the field 'event.name' is equal to the string '*Failed login attempt*'. Such a failed login attempt event may be generated by a monitoring software attached to a desktop machine or server and be forwarded by an attached connector. This aggregation rule is defined such that its action is executed when there are five such 'Failed login attempt' events within a time span of two minutes. The action of Rule 1 will generate a new event that has the name '*Suspicious login attempt*' and its *userid* field will be set to the userid value of the events that contributed to triggering this rule.

We note that userid is part of the aggregation condition and thus identical for all the five events that contributed to the action being executed. The second rule in Figure 1, '*Rule 2: System compromised'* is a join rule that correlates two events. The Event-1 condition checks for an event that is generated from Rule 1 and has a userid value that is found in a list of known inactive users. Event-2 checks for a successful login event. There may be many successful login events, however we are only interested in the ones that have the same userid as the '*Suspicious login attempt*' event. In Rule 2, this correlation condition between the two events is expressed using the join condition '*Event-1.userid = Event-2.userid'*. If the system finds an event that matches Event-1 conditions as well as an event that matches the Event-2 condition and the correlation condition is met, then the corresponding action will be executed.

Rule 2 in the above example also has a condition where a lookup in an internal *data list* is required as is expressed using the *'in'* clause. Data lists are dynamic tables that collect specified field values of event data. They serve as a community bulletin board for tracking specific event data over long periods (days or weeks) so it can be available on demand for lookups. E.g., we can define a 'hostile attempt' rule that places systems that show hostile activity into a data list called 'hostile systems'. Another rule may poll this 'hostile systems' data list to only consider events that stem from assets considered hostile.

A. Rules engine

As can be seen in the example rules above, rules take the form of 'If <conditions> Then <action>' statements. A rule is said to *be matched* or *fire* when the conditions of the rule become true. The SIEM system first needs to evaluate the conditions of a rule in order to identify if the rule can be fired. For simple rules this is straight forward. However, for correlation rules this is more complicated as participating events may arrive at different points in time. To facilitate this, information about the event and results from condition evaluation are maintained as facts in a *working memory*. A working memory is a specialized data structure for storing such facts. E.g., when one event with '*event.name = Failed Login attempt*' is processed, it *partially matches* the aggregation rule, Rule 1 in our example. This means that the current event may contribute to the rule being fired in future, but we cannot fire the rule yet. We refer to such facts using the term '*partial match info*'. Similar logic applies for join rules. While processing Rule 2, details of an event that matched Event-1 conditions may already be in the working memory when a new event arrives. We use the term *rules engine* to describe the set of algorithms and data structures that perform the rule matching. In short, the rules engine looks at the facts in the working memory and generates the list of correlation rules that can be fired. A naïve way to identify rules to be fired is to match each rule condition against all known facts in the working memory. Different rules engines use different algorithms and data structures offering different tradeoffs in terms of execution time and memory usage. Most of these algorithms are enhanced versions of the RETE algorithm [5].

B. Event flow

The flow of an event through a rule-based SIEM system is shown in Figure 2. Events arrive and are buffered in an event pool. The rules engine itself consists of multiple worker threads

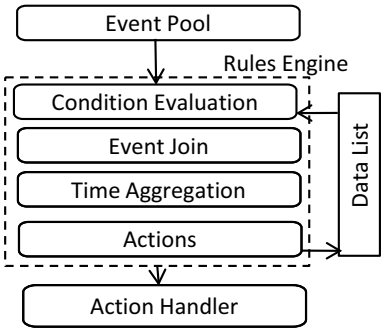

Figure 2: Event flow

that can process events in parallel. The steps of operations performed by the worker thread for each event are described below.

1. The worker thread picks up the next event from the event pool for processing.
2. For each rule, corresponding conditions are evaluated against the event field values.
3. If the event matches all conditions of a simple rule, then that rule's actions are triggered immediately. Simple rules do not produce any partial match info and hence no facts are inserted into the rules engine. The event may also partially match a join or aggregation rule. In such cases, the partial match info is inserted into the working memory of the rules engine. This step is performed for all the rules in the system.
4. After all partial match information is inserted, the rule matching algorithm in the rules engine is triggered. The rule matching algorithm goes through all existing and newly added facts in the working memory to identify rules that need to fire. If any matching rules are found, then the corresponding actions are triggered.

Figure 2 also shows how data lists are used. Actions triggered by simple, correlation or aggregation rules can make changes to the data lists by adding, updating or removing entries. These data lists can be queried during rule condition evaluation.

III. DRES Architecture

This section describes the design of DRES, our distributed rule evaluation system and its implementation.

A. Design challenges

We use the term *node* to refer to a physical or virtual machine on which a DRES instance runs. The main challenges of building DRES are described below:

1. A single node should not become a bottleneck when scaling up throughput.
2. An event may be processed by any node. Events that match an aggregation rule thus may be spread across instances on different nodes. E.g., if we consider Rule 1 from Figure 1, it is possible that two 'failed login attempt' events arrive at node *A* and three 'failed login attempt' events arrive at node *B*. If this happens, none of the two nodes by themselves can detect that there are five matching events in total. DRES

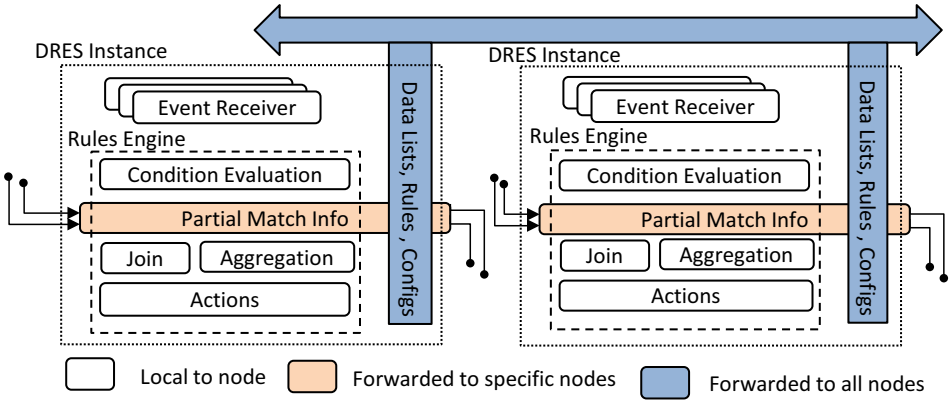

Figure 3: DRES Architecture

needs to ensure that such situations will be detected and the corresponding actions will be fired. The same holds true for join rules.

3. As explained in Section II, rule conditions can query dynamic data lists and rule actions may update data lists. This implies that an event being processed on node *A* may generate a data list update that affects how events are processed in node *B*.

B. Design

1) Overview

One key requirement of DRES was that it should provide the same functional behavior as an existing centralized single node rule evaluation system. As a first step, we evaluated the bottlenecks in such a single node solution. We found that condition evaluation of rules accounted for 80% of the total processing time that an event spends in the event flow. We note, that this is the case although the condition evaluation is already thread parallelized and heavily optimized (e.g., conditions that are part of multiple rules are only evaluated once and intermediate results are buffered).

Figure 3 shows a simplified high level architecture of DRES. Each dotted box represents the components of a DRES instance that runs in one node. Each node has its own event receivers, condition evaluation component, rules engine, and a component that fires actions. Rules and configurations are synchronized between nodes. We assume a high performance load distributor that deliver events from various connectors to DRES event pools. Event receivers in each DRES instance pick an event from the event pool and start processing it by performing condition evaluation. This works well because condition evaluation for an event is mostly independent from condition evaluation of other events. The only exception to this rule is data list lookups. To facilitate fast lookups, data lists are replicated in each DRES instance as described in Section III.B.3). This enables every node in a cluster to perform condition evaluation, the most time consuming process in the event flow.

Partial match information generated by one node may be processed by another node in the cluster. The exact semantics and details of how partial match info is forwarded to nodes is discussed in Section III.B.2). Each node processes the partial match info assigned to it by inserting the information into its rules engine. The rules engine checks for rule matches and triggers actions if matches are found. Simple rules can be completely processed locally on a node as these do not generate partial matches or make use of the rules engine. Actions from simple rules can be triggered locally as well. Further, actions of simple, join or aggregation rules can make updates to one or multiple data lists. Data lists are dynamic. DRES maintains its data lists in such a way that all nodes in the cluster see updates to the data lists in the same order. Details of how we handle updates to data lists are shown in Section III.B.3).

2) Partial match forwarding

If an event partially matches an aggregation or join condition then it follows the sequence of operations illustrated in Figure 4. Below we describe the sequence of operations that occur when an event *e* generates a partial match in DRES:

1. The node that is processing event *e* generates a list of rules that *e* partially matches.
2. For each rule in the list, DRES creates a hash value by considering a specific subset of event field values. For a join rule these are the fields that are part of the join equality condition. This means, if we have a join rule of the form <E1, E2, J> and an event *e* that produces a partial match by satisfying either of the E1 or E2 condition, then we consider the values of the fields that occur in the equality conditions in J to create the hash value. This ensures that for an event *e* all events that potentially can be joined with *e* will have an identical hash value. Similarly, for an aggregation rule we use all the identical fields over which events are aggregated to create the hash value. We use the term *partial match hash* to refer to this hash value.
3. The partial match hash value from the previous step is then converted to a node id (an integer between 0 and N-1, where N is the total number of nodes in the system) using a consistent hashing [6] function.
4. The partial match information for each rule is forwarded to the node with the corresponding node id. If for any rule the node id corresponds to the local node's id (the node that received and started processing event *e*), then the partial match information is inserted into the local rules engine directly. The size of partial match

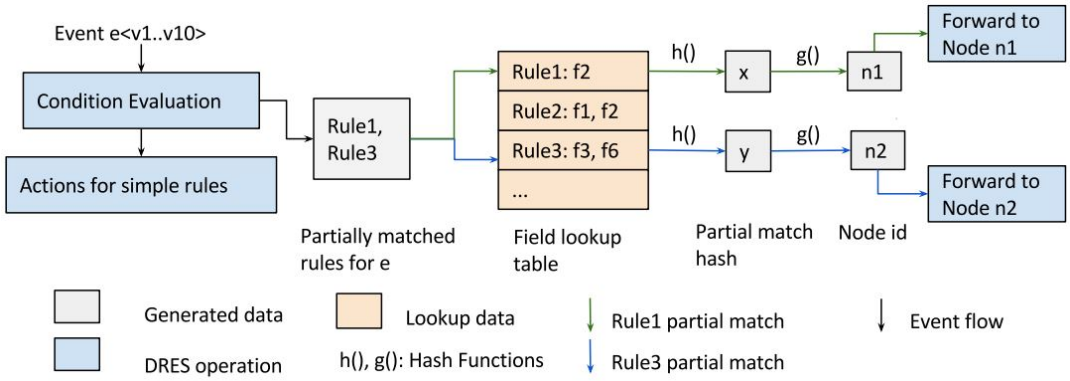

Figure 4: Distributed partial match forwarding

information is only a small fraction of the actual event size as received by the node. Thus, this approach minimizes the intra cluster communication, which is critical for scaling to a large number of nodes.

Figure 4 shows an example of the distributed partial match forwarding sequence for an event *e* that partially matches Rule1 and Rule3. The field lookup table in the figure lists the fields used in join equality conditions for join rules and identical aggregation fields for aggregation rules. Using this lookup table, DRES identifies the fields of e that must be considered for generating the partial match hash. Using the hash function *h()*, DRES then computes *h(e.v2)* for computing the partial match hash value *x* for Rule1 and *h(e.v3, e.v6)* for computing the partial match hash value *y* for Rule3. The partial match hash values are then converted to node ids using the consistent hashing function *g()*. Assuming *g(h(e.v2))* identifies node *n1*, the partial match info for Rule1 is forwarded to node *n1*.

The intuition behind this approach is that two events can match a join rule only if the values of the join condition fields are identical. Thus we can achieve correctness if we can maintain the invariant that one node will receive all partial match information of events that can be joined with each other, irrespective of which node processed the events. By creating a hash value of events per rule that is partially matched and based on all fields in the equality join conditions, the join partner events will have the same hash value. So, forwarding to a node based on the generated partial match hash value ensures this invariant. Similarly for aggregation rules, creating a hash value per partially matched rule and based on all fields in the aggregation clause ensures that events that can be aggregated together will have the same partial match hash value.

Once the partial match hash is created and the node id is identified, the event, the partial match info for the event, and the corresponding rule id is forwarded to the determined node. In the event flow, this happens after condition evaluation and generation of partial match results. This means that the node that initially received the event from the connector has already done 80% of the computation before forwarding it.

3) Data lists consistencys

Rule condition evaluation often requires data lists lookups. These frequent reads from data lists during condition evaluation heavily impact the overall system performance. Ensuring minimal latency for data list reads is critical for system performance. Maintaining data lists in a remote node (e.g., a centralized data list repository) would increase read latency and thus negatively affect performance. Further, writes to the data lists are orders of magnitude less frequent than reads. Thus, we maintain replicas of data lists within each DRES node, allowing very fast local lookups during condition evaluation. To ensure that DRES provides the same functional behavior as the previous single node system, the data list replicas are maintained eventually consistent using Paxos [7]. Paxos is a protocol used to achieve consensus among multiple nodes. Consensus refers to the process by which all nodes in the system agree on one result. In DRES all nodes agree on the order in which updates are applied to data lists. The initial state of the data lists in all DRES instances after system initialization is identical. This means, if the order in which updates are applied to this initial state is the same across nodes, then the data list state will be eventually consistent across all nodes.

4) Scalability, portability

The main advantage of our system is the ability to scale out as load (measured in events per second, EPS) increases. Distributing the compute intensive condition evaluation, event correlation and aggregation enables linear scaling of the total event throughput with the number of available nodes in the cluster up to clusters of tens of nodes as shown in Section IV. The partial match forwarding technique outlined above has the advantage that there are no changes necessary to the rule correlation engine itself. The actual code for aggregation and joins remains unchanged regardless of the mode of operation (single node or distributed). This is an important factor for legacy reasons as well. Further there is no requirement to distribute the set of rules across the nodes as all nodes process all rules.

C. Implementation

This section describes how we implemented the distributed architecture described above. We built DRES by modifying the existing centralized, single node SIEM system, which is implemented in Java. For generating the partial match hash values and consistent hashing we use the Guava library [8]. For forwarding the partial match messages, we implemented a messaging system using ZMQ [9]. ZMQ is a library based,

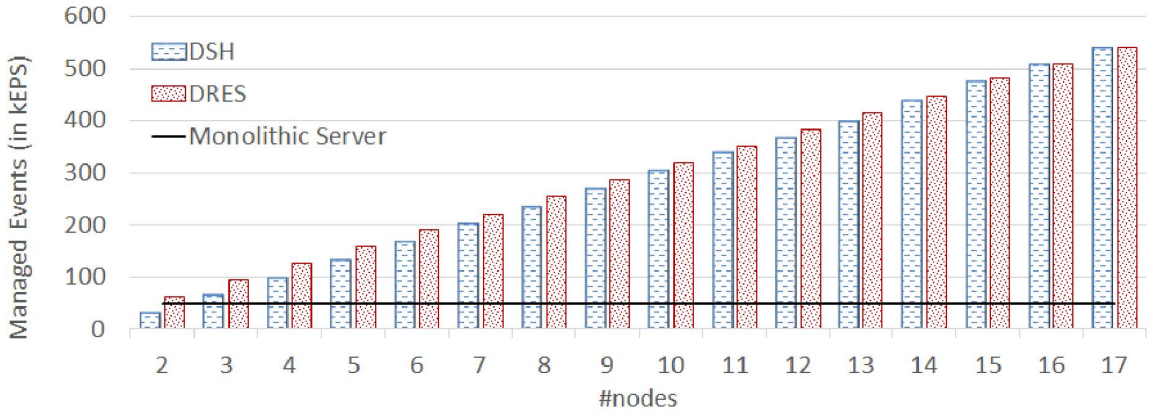

Figure 5: Scalability under normal trace

broker less transport layer messaging framework for distributed applications. ZMQ sockets are initialized when DRES starts up and kept alive throughout the life of the application. Each node maintains one pull socket that reads incoming partial matches from all the other nodes. A node also maintains a push socket for every other node to which partial match information is forwarded. To reduce the latency overhead caused by serializing and deserializing the outgoing and incoming partial match information messages, respectively, we use the Kryo [10] serialization library. Once the condition evaluation component identifies a partial match, the event and the corresponding partial match information are serialized and inserted into the send queue. A worker thread picks up the serialized byte array and sends it to the destination node using the ZMQ push sockets. One event may partially match multiple rules and information for each partial match may need to be forwarded to a different node.

For maintaining data list consistency, we use the JPaxos [11] implementation of Paxos. JPaxos provides a simple and efficient Java library for Paxos replication. Each DRES instance maintains a local Paxos thread that propagates data list updates. Data list updates determined by the protocol are picked up by a separate thread running on each DRES instance. This thread then executes these updates to the local data list.

D. Discussion

1) Rule partitioning

With respect to distributing computation, another option that we explored was partitioning the set of rules among nodes. This would address two of the three design challenges mentioned in Section III.A). We decided against it for the following reasons:

- Expert knowledge is required to distribute a rule set into multiple smaller rule sets. As events can match multiple rules it is difficult to determine smaller rule sets without the need of duplicating too many events.
- Additional logic is required at the connectors to identify the nodes to which an event needs to be forwarded.
- The number of matches per rule is not uniform. We have use cases where at times one or two rules are responsible for a vast majority of the join or aggregation partial matches. This will result in a large number of underutilized nodes and few over utilized nodes.
- Scaling the number of nodes becomes difficult as it requires changing logic at the connectors as well as re-partitioning the set of rules.

2) Fault tolerance

Our system design also ensures fault tolerance as discussed below.

- Since we use consistent hashing ([6], [12], [13]) to identify the nodes to which partial match information is forwarded, in case of a failure, we can redistribute the partial match forwarding targets among the remaining nodes with comparatively little disruption. The target node for a partial match is identified based on the partial match hash value. In case of a naïve scheme to identify the target node (e.g., hash value mod N), if a node is added or removed, the target node corresponding to each partial match hash value may change and require remapping. Consistent hashing ensures that only $1/N^{th}$ of the total possible hash values require remapping for a cluster of N nodes.
- Using Paxos for replicating data list updates across all nodes also allows us to tolerate node failures. Paxos enables the system to make progress as long as a majority of the nodes are alive. This ensures progress even when some nodes fail. Paxos also allows failed nodes to catch up on updates from persistent logs during recovery.

IV. EVALUATION

This section evaluates the performance of DRES compared to other solutions. In detail we evaluate (i) scalability of DRES under normal and heavy workload, (ii) network utilization of DRES and (iii) the effect of forwarding partial matches.

A. Experiment Setup

Our experiments were run on HP SL servers, each having two 6-core 2.66 GHz Intel Xeon X5650 CPUs (effectively 24 cores with Hyper-Threading enabled), 96GB of DRAM and a 128GB SSD. All servers are connected via 10Gbit Ethernet NICs and we use a dedicated server per DRES instance. To generate client traffic we use a tool that generates a workload by

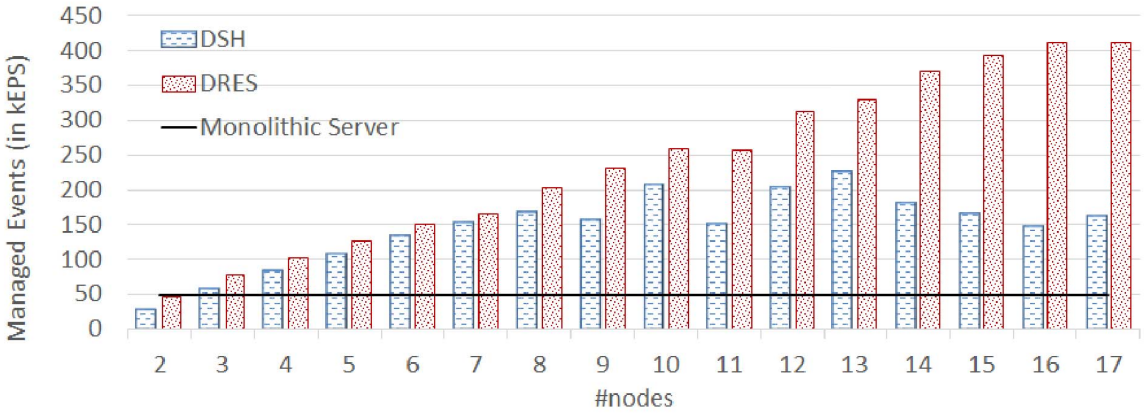

Figure 6: Scalability under heavy trace

creating a stochastic stream of events from an input trace file. We use two different traces. Both represent real workloads reflecting a system at periods with many security breaches. These workloads are replayed to benchmark a real and widely used enterprise security management product. For the experiments in this work, we use one workload generator for each node in the DRES cluster. This removes the potential bottleneck of a load distribution component and lets us focus on the performance of DRES itself. We note that the workload generators are running on a different set of servers than the system under test. The generators adjust their workload and communicate with the SIEM system (consider feedback) to determine the maximum event rate that the system under test can handle. After a short warm-up phase, the workload generators produce load for 20 minutes. After 20 minutes, the workload generators terminate and report the average sustained event rate expressed in events per second (EPS) that they were able to send to their DRES node without overloading it. We then sum up the event rates of each generator to determine the total event rate of the DRES system.

TABLE 1. WORKLOAD CHARACTERSITIC

	Normal trace	Heavy trace
Join matches	29%	84%
Aggregation matches	4%	50%
Rules fired (per 1000 events)	1.26	1.34
State updates (per 1000 events)	0.19	0.32

We present results for two different workload traces; normal trace and heavy trace. TABLE **1** describes the nature of the two workload traces. In the table, '*Join matches'* refers to the number of partial matches generated by events matching join rules in relation to the total number of events. Similar, '*aggregation matches'* refers to the number of partial matches generated by events matching aggregation rules in relation to the total number of events. '*Rules fired (per 1000 events)'* refers to the average number of rules that are completely matched (and actions executed) when 1000 events from the trace are processed. '*State updates (per 1000 events)*' refers to the average number of data list updates triggered by the workload. State updates are triggered by a subset of the actions that are executed. '*Normal trace*' represents a typical client load with respect to the number of join and aggregation matches and actions triggered. '*Heavy trace*' is intended to further stress test the system by heavily increasing the number of events that partially match a join or aggregation rule.

Both the normal and heavy event traces are executed against a real production rule set used in active deployments. This rule set contains 167 rules. Rule conditions contain a total of 153 instances of lookups from one or more data lists. There are also a total of 91 unique instances of rule actions updating a data list. 46 of the rules contain aggregation components that group more than 1 event in a sliding window and 27 rules contain join conditions. At a more fine grained level, the rule set contains a total of 863 *AND* conditions and 308 *OR* conditions.

B. Evaluated SIEM Systems

This section describes the three different systems used for evaluation. Rules and relevant configurations remain the same in all three systems described below during the evaluation.

1) Monolithic server

This refers to the standalone, single server version of our SIEM system. The SIEM system uses the same rules engine and has the same event flow and functionality of DRES.

2) Distrubuted single handler (DSH)

In order to evaluate the gains of a fully distributed version, we also built a distributed single handler (DSH) version of the rules evaluation engine. In DSH, we designate one node as the handler node and rest as worker nodes. The connectors forward events to any of the worker nodes. Each worker node then performs condition evaluation and forwards all generated partial match info to the designated handler node. Again, simple rules are evaluated locally. The handler node inserts the received partial match information into its rules engine and checks for rule matches. This means, all join and aggregation rule actions will be triggered at the designated handler node. We note that data list updates still can be triggered by all nodes in the system as simple rule actions are triggered directly after condition evaluation at any worker node. Updates to data lists are forwarded to the handler node. The handler node receives all

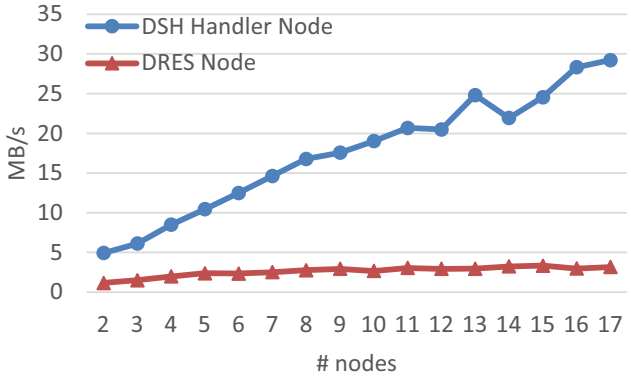

Figure 7: Network load under normal trace

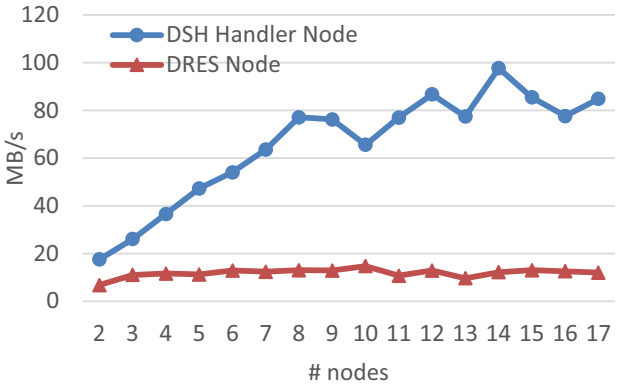

Figure 8: Network load under heavy trace

data list updates, orders them, applies them locally and then broadcasts the ordered updates to all workers. The worker nodes then apply them locally to their data list replicas. As with Paxos, this ensures that the order of data list updates remains the identical on all nodes ensuring the same eventual consistency guarantee as offered by DRES.

3) Distributed rule evaluation system (DRES)

This refers to our fully distributed rule evaluation system as presented in Section III.

C. Scalability

In order to study the scalability of our solution, we evaluate how well total system throughput scales with the number of nodes. We note that for DSH deployments one node is a dedicated handler node, i.e., there are a total of *#nodes-1* worker nodes available that accept events from the clients. For DRES deployments, all nodes accept incoming events. Figure 5 shows the total system throughput against the normal trace and Figure 6 shows the throughput against the heavy trace. In both figures, the black horizontal line represents the EPS of the standalone server which is around 49K. As can be observed from Figure 5, both DSH and DRES scale well under the normal workload, crossing half a million EPS with 16 nodes. We note that this is more than ten times the throughput of the highly optimized monolithic server (49K) installation. Already with two nodes, DRES supports more EPS than the monolithic server. Under the normal workload DSH is also able to scale well.

Figure 6 shows how the systems perform under the heavy trace. DRES still scales linearly because all nodes perform aggregations and joins and thus distribute the load among each other. As can be observed, DSH throughput peaks at around 10 nodes. The heavy workload trace generates a much higher number of join matches than the normal workload resulting in a higher number of partial matches being forwarded to the handler node in DSH. For larger cluster sizes the central handler node in DSH requires a very high amount of DRAM to maintain all partial match info in the correlation engine. This saturates the handler and limits overall system performance.

D. Network Load

Figure 7 and Figure 8 compare the network load of the DSH handler node with the network load of one node in the DRES cluster under both, the normal and the heavy trace, respectively. We note that all nodes in the DRES cluster exhibit a very similar network traffic. As expected for DSH, network load increases linearly for the DSH handler node with the number of nodes in the cluster until the handler node gets saturated. This stems from a single node performing all join and aggregation processing. As the number of nodes increases, the DSH handler has to deal with receiving partial match information from more nodes. In contrast, the DRES cluster shows no significant increase in network load at any node in the cluster as all nodes perform join and aggregations and thus the load is shared between them.

E. Partial Match Forwarding

Figure 9 compares the number of partial matches forwarded in DSH and DRES under the normal trace. For the DSH deployment, forwarded matches represents the sum of partial matches forwarded by each node to the central handler node. For DRES, forwarded matches represents the sum the all partial matches that are *not* handled locally by nodes. Figure 10 shows the same comparison for the heavy workload trace. In both evaluations, we can observe that the total number of events forwarded by DSH and DRES are similar. For DRES, the number of partial matches forwarded to other nodes is expected to be 1/N smaller than in DSH (with N being the number of nodes) due to local processing of correlations. However in most experiments this reduction is offset by the slightly higher EPS of DRES. Further, for the heavy load trace, the total number of forwarded partial matches of DSH drops for clusters larger than eight nodes. This results from the central handler node being overloaded and thus limiting overall system throughput.

F. Summary

We showed that DRES scales linearly to at least 17 nodes supporting much more than half a million events per second. This is over 11x increase over the monolithic, standalone server. The results also showed that our system could scale up well beyond the DSH deployment under heavy workload. Network load on the DRES nodes remain stable under both workloads showing promise of even higher scaling opportunities.

V. Related Work

This section summarizes related work on SIEM systems and other event monitoring and analysis frameworks.

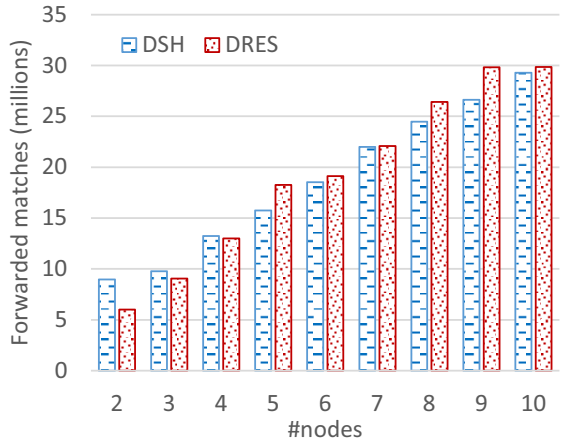

Figure 9: #Partial matches forwarded under normal trace

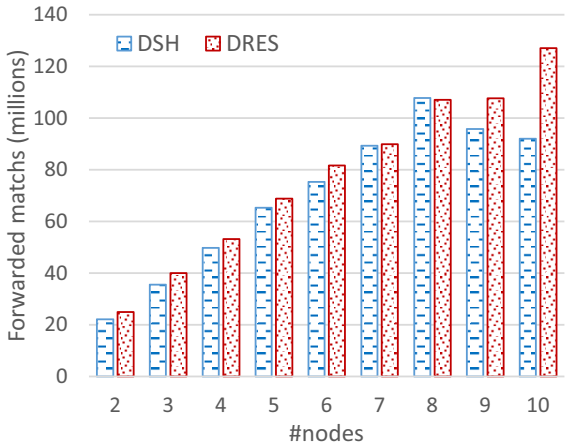

Figure 10: #Partial matches forwarded under heavy trace

A. *Infrastructure monitoring*

One of the tasks done by SIEM systems is to perform infrastructure monitoring. Some of the most popular and widely used systems that perform distributed infrastructure monitoring are Ganglia [14] and Nagios [15]. Ganglia is a distributed monitoring system for compute clusters and scales to large clusters using a hierarchical design. Nagios is an IT infrastructure monitoring tool that can monitor network communication, disk usage and other parameters of the cluster. Though good for monitoring cluster metrics, neither system supports rule-based knowledge inference, event joins and actions based on rules. Rule defining support in Nagios is basic and is limited to generating alerts when systems fail. We note that both, Ganglia and Nagios, can be used to collect data, which is then forwarded to DRES for further processing.

B. *Big data analysis*

There are two primary approaches to processing Big Data: batch processing and stream processing. In batch processing, data is at rest, finite and known before the computation starts. Whereas in stream processing, streams are assumed to be infinite and stream processing systems report results as soon as they are ready. Examples of popular batch processing systems include MapReduce [16], Dryad [17] and various languages implemented on top like Pig Latin [18]. The Hadoop Online Prototype [19] [20] makes MapReduce 'online', allowing it to support continuous queries, which enable programs to be written for applications such as event monitoring and stream processing. Themis [21] further improves the performance of I/O bound MapReduce jobs by reducing the amount of disk writes. Many popular data stream processing systems can also perform joins and aggregations over event streams. Apache Storm [3], provides a generic computation framework that allows programmers to define topologies that perform computation over data streams. Programmers can develop topologies that aggregate events in a specified time interval or perform arbitrary computation. Schneider *et al*. present a compiler and runtime system [22] [23] that automatically extract data parallelism for distributing stateful streaming applications. Apache Spark [4] allows big data analytics over distributed, in-memory data sets. The Muppet [24] system allows programmers to process stream data by proving a map and update interface. The update function can maintain state across the system by writing to a storage data structure called slate. Though these systems are capable of event processing, their primary focus is to provide a generic computation framework for streaming data. Event evaluation systems like DRES, focus on providing a non-programmatic rule-based interface to the end user. This means, the end user simply provides a set of predicates and actions, and the complex task of matching rules against predicates falls to DRES.

C. *Distributed event correlation*

IBM's System S/Infosphere Streams [25] perhaps represents the state-of-the-art in distributed stream processing. System S offers an analytic platform that allows user-developed applications to analyze and correlate information as it arrives, in real-time. System S also uses an operator-based programming language called SPADE [26]. Similar to System S, Esper [25] also allows complex event processing and can scale by executing its operators inside Spark or Storm executors [27]. However, both System S and Esper do not support dynamically varying patterns that depend on shared state. StreamSQL+Streambase is another stream processing language but is not distributed and does not support dynamic patterns. Event correlation has also been explored in the context of content based publish-subscribe systems. In the publish-subscribe model, information flow into the system as events or messages that come from multiple publishers. Such systems (e.g., SIENA [28] and Gryphon [29]) rely on a broker network to deliver events to subscribers and networks typically rely on advertisements issued by subscribers or publishers to create a routing network to forward events. Rules engines like Jess [30] and Drools [31], support dynamic patterns but are neither scalable nor distributed, and have well-known performance problems when the heap size of the associated Java Virtual Machine (JVM) is increased beyond 2GB. The DCEP complex event processing system developed as part of the PLAY project [32] [33] also aims to provide a cloud based middleware platform to combine heterogeneous events in a distributed fashion. DCEP relies on a distributed event store to query events but the event evaluation itself is done by a single component. Our implementation reveals opportunities for different and

perhaps complementary optimizations, such as distribution of event ingest, aggregation and correlation.

Other recent approaches to scaling complex event processing include RIP [34], which tries to identify patterns in event streams in order to distribute input events that belong to individual run instances of a pattern to different nodes, thereby providing fine-grained partitioned data parallelism. RIP applies this approach to multi core processors and does not explore distribution over the network. Brenna *et al*. [35] also implemented a distributed event pattern matching system on top of Cayuga [36] and Spread [37]. They try to scale by partitioning the queries and running multiple queries in multiple machines in parallel. This suffers from similar drawbacks as the rule partitioning approach that we discussed in Section III.D.1).

VI. Conclusion and future work

In this paper we presented DRES, a distributed rule evaluation and event management system for enterprise level monitoring. This system enables real-time processing of the ever increasing number of arriving events. Using various production test workloads, we demonstrate that it scales very well and can perform aggregations and joins on more than half a million events per second using a cluster of 17 nodes. This is 11 times more than any comparable single node system can achieve today. We outline the design goals, architecture and implementation of the system and show how the system performs under a normal and heavy workload trace.

In our current implementation we focus on the correlations between two events. For the next version, we plan to support correlations between multiple events. This can be achieved by forwarding intermediate correlation results between instances using the presented hashing technique for partial match information. Further, we are investigating techniques for DRES to handle node failures automatically. This presents two orthogonal challenges: First, for no-loss recovery we need to be able to recover the rule state in the failed node from persistent storage, backup processes etc., or an ability to replay parts of the event queue. Second, we need to be able to reassign the processing done by the failed node to the other operational nodes.

References

[1] "Gartner," *Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020*, 12-Dec-2013. [Online]. Available: http://www.gartner.com/newsroom/id/2636073.

[2] "Hadoop." [Online]. Available: http://hadoop.apache.org/.

[3] "Apache Storm." [Online]. Available: https://storm.incubator.apache.org/.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In HotCloud '10.

[5] C. L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell.*, vol. 19, no. 1.

[6] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In ToC '97.

[7] L. Lamport. The Part-time Parliament. *ACM Trans Comput Syst*, '98.

[8] "Guava." [Online]. Available: https://github.com/google/guava.

[9] "ZMQ." [Online]. Available: http://zeromq.org/.

[10] "Kryo." [Online]. Available: https://github.com/EsotericSoftware/kryo.

[11] "JPaxos." [Online]. Available: https://github.com/JPaxos/JPaxos.

[12] A. Dury. Peer-to-Peer Computing in Distributed Hash Table Models Using a Consistent Hashing Extension for Access-Intensive Keys. In AP2PC '04

[13] G. Swart, Spreading the Load Using Consistent Hashing: A Preliminary Report. In PDC '04.

[14] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. In Parallel Comput., vol. 30, no. 7. '04.

[15] "Nagios." [Online]. Available: http://www.nagios.org/.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI '04.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In EuroSys '07..

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In SIGMOD '08.

[19] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online Aggregation and Continuous Query Support in MapReduce. In SIGMOD '10.

[20] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In NSDI '10.

[21] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: an I/O-efficient MapReduce. In SOCC '12.

[22] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing Stateful Distributed Streaming Applications. In PACT '12.

[23] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic Scaling of Data Parallel Operators in Stream Processing. In PDP '09.

[24] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-style Processing of Fast Data. In VLDB '12.

[25] "IBM Infosphere." [Online]. Available: http://www-03.ibm.com/software/products/en/infosphere-streams.

[26] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System s Declarative Stream Processing Engine. In SIGMOD '08.

[27] N. Zygouras, N. Zacheilas, V. Kalogeraki, D. Kinane, and D. Gunopulos. Insights on a Scalable and Dynamic Traffic Management System. In EDBT '15.

[28] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans Comput Syst*, vol. 19, no. 3, pp. 332–383, 2001.

[29] R. E. Strom, G. Banavar, T. D. Chandra, M. A. Kaplan, K. Miller, B. Mukherjee, D. C. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In CoRR '98.

[30] "Jess Rules Engine." [Online]. Available: http://www.jessrules.com/.

[31] P. Browne, *JBoss Drools Business Rules*. Packt Publishing, 2009.

[32] R. St¨uhmer, Y. Verginadis, I. Alshabani, T. Morsellino, and A. Antonio. PLAY: Semantics-based Event Marketplace. In Virtual Enterprises '13.

[33] N. D. Stojanovic, L. Stojanovic, and R. Stuehmer. Tutorial: Personal Big Data Management in the Cyber-physical Systems - the Role of Event Processing. DEBS '13.

[34] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing. In DEBS '13.

[35] L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In DEBS '09.

[36] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: A High-performance Event Processing Engine. In SIGMOD '07.

[37] Y. Amir, C. Danilov, and S. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In DSN '00.