Abstract—The explosion in consumer devices has resulted in a significant increase in the number of mobile telecommunications nodes. As a result of increased device and node numbers, network operators have experienced a large increase in associated events. In such an environment, scalability and performance of event handling become important aspects for Operation Support Systems (OSS). A traditional approach has been to centralize monitoring and decision functions. The scale of events in a modern mobile telecommunications network means such centralized implementations are performance limited. What is required is a remodeling of Complex Event Processing (monitoring) and Policies (decision making) towards a distributed yet coordinated system. This paper describes an extensible architecture for such a distributed policy-based event processing system. Our approach provides a pluggable mechanism into which various event handling functionality can be integrated. In order to illustrate the applicability of our approach we evaluate the performance of 2 message queuing protocols; Advanced Message Queuing Protocol (AMQP) based RabbitMQ and Java WebSockets. Our performance evaluation illustrates the ability of our architecture to transparently integrate alternative event processing technologies.

Keywords—Complex Event Processing, Rule System, Distributed Processing, Performance

I. INTRODUCTION

Ever increasing number of mobile network events due to increased number of mobile devices, demands a scalable architecture. Our work combines rule systems to encode event processing knowledge and messaging to provide for a distributed system. Other earlier work used centralized rules over a distributed system [7-10], which drastically limits the scalability. The main advantage of our approach is that policies used are (a) distributed (over multiple components) and (b) coordinated (using centralized authoring). In this paper, we describe the extensible architecture, the reference implementation and performance tests with regard to end-to-end message processing.

This paper is organized to provide a brief reference in Section II to the related work in the domain. Section III introduces core concepts, technologies and products from messaging and rule systems. Section IV elaborates the extensible architecture and gives detailed description of it’s various components. Further in Section V, we discuss the implementation of those components (we call them nodes, 2 core nodes and 2 supporting nodes). In Section VI, we discuss the results obtained from several test runs through the system and compare two off-the-shelf business messaging protocols, Advanced Message Queuing Protocol (AMQP) based RabbitMQ and Java WebSockets.

II. RELATED WORK

In the “Policy-Based Information Sharing in Publish/Subscribe Middleware” [7] author describes a control of sensitive information system in health care environment. The criticality of information sharing and data access is controlled by rules, precisely hook rules (Postgres SQL). Information that travels on the messaging system is tailored for a particular subscriber, on need-to-know basis. We have found that this paper has similar architecture as of our system with a slightly different implementation. Our system analyzes the patterns inside the incoming messages and modifies the forwarded message to correspond to the identified pattern, whereas it analyses the incoming messages and modifies it for particular subscriber according to information relevant to that subscriber.

“A rule-based middleware for business process execution” [8] implements rules over messaging middleware to provide a simple and efficient way of describing executable business processes. The complex conditional workflows and enterprise integration patterns are implemented in terms of rules. The Prova rule language and the Rule Markup Language (RuleML) are used to implement rules over an Enterprise Service Bus (ESB).

“Policy-driven middleware for self-adaptation of web services compositions” [9] focuses on specifying and enforcing monitoring-policies to help in fault detection and corrective adaptation of web services compositions. Since monitoring and corrective action selection is combined in a single policy, this work does not scale well when the number of faults increases drastically. It also does not allow for smart filtering of fault events, which is essential to address high-priority events immediately and add lower-priority events to maintenance reports.

“Message oriented middleware with integrated rules engine” [10] is a patented invention addressing deficiencies...
in respect to the management of message oriented middleware. It describes the integration of a rule engine with message-oriented middleware. Their method includes creating a shared memory in the memory of a computer and adding or deleting tokens in the shared memory corresponding to objects such as messages and message queues, created in and removed from, respectively, in a messaging component of message oriented middleware, or topics or subscriptions or log file space for messages queues in the messaging component. The method additionally includes applying rules in a rule engine to the tokens in the shared memory.

Our work differs from the above in that we use distributed and coordinated policies (between two components for event processing and governance), while policy instances in each component are atomic, i.e. do not effect each other. This results in a system that is hugely scalable, since only a combination of event processing policy and governance policy depend on each other.

III. CONCEPTUAL BACKGROUND, PRODUCTS AND TOOLS

Combining concepts from messaging systems with concepts from rule systems requires an understanding of two disjoint domains. In general, messaging system provides the main communication links between the components of a distributed system. A rule system provides the intelligence to manage and process events and event patterns to trigger appropriate actions. In this section we look into the fundamental idea of both to introduce relevant terms and concepts.

A. Messaging System

A distributed system has multiple components that may be built independently, with potentially different languages and platforms, dispersed at different locations. There are a number of approaches including: distributed data stores, streamed data, query-response models, or asynchronous messaging. Using a message-based approach distributed components share and process data in a responsive asynchronous way and it is this approach we focus on in this work. Our system architecture has a pluggable design to integrate with different messaging systems. We started with AMQP messaging due to external project requirements and later inserted Java-WebSockets into our system and did a performance evaluation for comparison between the two.

1) RabbitMQ: AMQP is “an open standard for passing business messages between applications” [1]. Data is sent in a stream of octets (or bytes), thus it is often called a ‘wire protocol’. Version 1.0 of the AMQP standard defines three main components: the networking protocol, a message representation and the semantics of broker services. All of these components address core features such as queuing, routing, reliability and security. Message encoding is separated into links, sessions, channels and connections, with links being the highest level and connections the lowest level of abstraction. A link connects network nodes, also known as distributed nodes in AMQP.

RabbitMQ [2] is an open source implementation of the AMQP standard. It facilitates ‘producers’ to send messages to ‘brokers’, which in turn deliver them to ‘consumers’. Messages can also be routed, buffered and made persistent, depending on runtime configuration.

AMQP is designed to be programmable, allowing application to configure ‘entities’ and ‘routing schemas’. The three important entities in RabbitMQ realizing the programmability are ‘exchange’, ‘queue’ and ‘binding’. An exchange receives events from a producer and realizes different routing schemes. A queue is bound to an exchange and handles consumer-specific message reception. A binding defines the rules for message transfer between an exchange and a queue. See [3] for details.

2) Java-WebSockets: Java-WebSockets [4] is an open source implementation of WebSocket protocol, in java. Data travels as Strings between server and clients. Clients only talk to server and sends data to server, server then broadcasts the data to all registered clients. See [4] for implementation details.

“The WebSocket Protocol is designed to supersede existing bidirectional communication technologies that use HTTP as a transport layer to benefit from existing infrastructure (proxies, filtering, authentication). Such technologies were implemented as trade-offs between efficiency and reliability because HTTP was not initially meant to be used for bidirectional communication. The WebSocket Protocol attempts to address the goals of existing bidirectional HTTP technologies in the context of the existing HTTP infrastructure; as such, it is designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries, even if this implies some complexity specific to the current environment. However, the design does not limit WebSocket to HTTP, and future implementations could use a simpler handshake over a dedicated port without reinventing the entire protocol. This last point is important because the traffic patterns of interactive messaging do not closely match standard HTTP traffic and can induce unusual loads on some components.” [5]

B. Rule System

Rule systems provide the means to define and process rules. In our work, we are focusing on Production Rule Systems (PRS) due to external project requirements. The computational model of PRS implements the notion of a set of rules, where each rule has a sensory precondition (“left-hand-side”, LHS, or “WHEN” clause) and a consequential action (“right-hand-side”, RHS, or “THEN” clause). Rules are also referred to as productions and they are the primary form of knowledge representation. The rule engine also maintains knowledge-base of facts. When the facts stored satisfy the precondition of a rule, the rule “fires”, thus invoking the action part of the rule. Often, the action part of the rule can change the fact knowledge-base, potentially triggering more rules.

Drools Expert is an open source implementation of a PRS. In Drools Expert, Rules and facts of a PRS constitute a
knowledge base. Rules are present in the production memory and the facts are kept in a database called working memory, which maintains current system knowledge. There is an Inference Engine based on Charles Forgy’s Rete Algorithm, which efficiently matches the facts from working memory to conditions of the rules in the production memory.

Also, a conflict resolution is required when there are multiple rules on the agenda. As firing a rule may have side effects on working memory, the rule engine needs to know in what order the rules should fire (for instance, firing ‘ruleA’ may cause ‘ruleB’ to be removed from the agenda). The default conflict resolution strategies employed by Drools Expert are: Salience and LIFO (last in, first out). [6]

IV. ARCHITECTURE AND DESIGN

We receive events from streams, process them and forward them via queues. Each component employs a rule engine to process events. A typical process is to receive an event or a number of events (pattern) and create/send composite events. The events we process are actual mobile network events, such as performance events (counters) or alarm events. However, for simplification we refer to events as characters, e.g. ‘A’, ‘B’ and ‘C’. Figure 1 shows how an incoming event stream (ABABCA…) is directed to a dedicated queue and processed.

Events are received, one by one, by the Complex Event Processing (CEP) component. It takes simple events (‘A’, ‘B’) and generates complex events (‘@A’, ‘@AA’). These complex events represent patterns, i.e. sequences of events that are of special interest. The rules in the CEP component specify which patterns need to be matched and which corresponding complex event needs to be generated. Finally, complex events are sent to the next queue.

The Distributed Governance (DG) component receives complex events and selects appropriate actions to respond to them. The rules in the DG component define which complex events are being processed and what actions are associated with them. The number of associated actions can be zero or more, with zero action indicating an undecidable situation, while more than one indicates multiple possible actions. DG then sends the actions to a new queue, which can feed into multiple applications of a broader management process, e.g. as part of Network Operation Center (NOC).

Combining messaging (AMQP) and rule systems (PRS) allows for a design of a flexible and scalable system. Using queues for communication not only facilitates the CEP and DG components to be distributed, but also for multiple redundant or load-balanced instances of each component to be run in parallel at runtime. If one CEP instance reaches its performance limits a new CEP instance can be executed, connected to the CEP queue and some patterns of the original CEP instance allocated to the new CEP instance. Figure 1 shows a scenario with three CEP instances and two DG instances.

One characteristic of the described system design requires special attention: the processing of patterns and the selection of actions is (a) distributed over two components (CEP and DG in the architecture) and can also be (b) distributed over multiple instances (CEP and DG instances in design and runtime). An effective and efficient coordination is required to guarantee that all patterns are processed and that the resulting complex events find related rules for action selection. Figure 1 shows a process for ‘Rule Authoring’ which is responsible for the coordination. The details of this process are out of scope for this paper, which focuses on the implementation and testing of the message processing.

V. IMPLEMENTATION

This section details the implemented system. We have built four components (which we call nodes), developed in Java 7. Two nodes realize the core of the event processing and two are used to automate tests. The two core nodes are CEP and DG (Figure 2). The other two supporting nodes are the input and output consoles (Figure 3). CEP and DG are built in a very similar way: they read events (messages) from a topic, invoke a rule engine to process events and then publish the results of the rule evaluation on another topic in form of complex events (CEP) or actions (DG).

A. Core Nodes

Both nodes, CEP and DG, start with an initialization of their respective topics and knowledge base (rules, for rule processing). CEP waits to get events from the input console, processes it (applies rules) and sends it out on another topic where DG receives it. Similarly, DG dispatches events with the associated action after processing the received composite event from CEP. This cycle of waiting and processing goes on endlessly for the core nodes.

Figure 1. Architecture and deployment scenario.

Figure 2. Core Nodes, CEP (left) and DG (right).
1) Complex Event Processing (CEP) Node: Figure 2 (left) shows the CEP node with its three main parts- start, wait and processing. Start creates the knowledge base and two topics CEP and DG. When an event is received on CEP topic, a corresponding fact is inserted into the knowledge base and all rules are ‘fired’ (processed). Rules evaluate to match patterns as the knowledge base holds the information (facts) of previously received events.

To keep the knowledge base light and efficient these facts are retracted when they are of no use to match patterns. In our system we have kept up to four facts in knowledge base to match the pattern, we call it the window of events. This window size can be changed per event pattern required to be matched. After rules evaluation complex events are generated and published to the DG topic.

2) Distributed Governance (DG) Node: Figure 2 (right) shows the DG node with its three main parts- start, wait and processing. Similar to the CEP node, DG creates its knowledge base and two topics called DG and OUT. The topic DG is the same as that created by the CEP node for its output, thus the two nodes a bound via that topic. When a complex event is received, a corresponding fact is inserted into the knowledge base and all appropriate triggered rules are then fired. Rules evaluate in DG to associate identified patterns to actions, which are then published to OUT topic.

<table>
<thead>
<tr>
<th>Single Event Patterns</th>
<th>Event</th>
<th>Composite Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>@A</td>
<td>Action-A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>@B</td>
<td>Action-B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>@C</td>
<td>Action-C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>@D</td>
<td>Action-D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>@E</td>
<td>Action-E</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multi Event Patterns</th>
<th>Event</th>
<th>Composite Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-A</td>
<td>@AA</td>
<td>Action-AA</td>
<td></td>
</tr>
<tr>
<td>A-B</td>
<td>@AB</td>
<td>Action-AB</td>
<td></td>
</tr>
<tr>
<td>A-A-B</td>
<td>@AAB</td>
<td>Action-AAB</td>
<td></td>
</tr>
<tr>
<td>A-A-B</td>
<td>@AABB</td>
<td>Action-AABB</td>
<td></td>
</tr>
</tbody>
</table>

* A-B implies Event ‘B’ comes after a previous ‘A’ event.

B. Supporting Nodes

For testing, we have added an input and an output console, which will later be replaced by real systems for event processing and action respond. For the current system the input node provides the functionality of reading a file containing events, and then splitting the string to publish events on the CEP topic one by one. The output node receives the actions on the OUT topic and prints them out.

Figure 3 shows the two supporting nodes and their main phases (input console on the left and output console on the right). The input console starts and publishes as described above. When all events read from a file are published, it terminates. The output console starts once and waits indefinitely (until the process is terminated). Start creates the topic OUT and wait waits for actions from the DG node to print them to the console as they arrive.

VI. TESTING AND EVALUATION

The tests we have performed are modeled to provide a good understanding about the performance of the overall system. Special attention focuses on the impact the message processing and the rule processing have on the overall system performance. The goal is to understand the technology impact on an end-to-end event processing. Tests have been run for 1 up to 50k events in a single stream with 10 test runs per input stream size. Event type used are java Maps (HashMap<String, Object>). The numbers of rules and the actual rules have not been changed between test runs, so the results show the processing of a fixed set of 10 rules for CEP and 9 rules for DG. Further test runs will be needed to understand the impact of increasing rule sets on the performance. All tests have been run on an Intel i5 (dual core) Windows 7 laptop.

Each component of the system has fixed measurement points. They are shown in the figures in the implementation section. Initialization phases (called start) are not part of the measurement. The following list shows all measurement points of each component:

- Core nodes (Figure 2): Start, Wait, Rules, Publish (for CEP and DG).
- Supporting Nodes (Figure 3: Publish (Input node) and Wait (Output node)).

These different time consumptions calculations are very important for the performance evaluation of a particular product. Our system architecture is designed such that it can work with different messaging systems and rule systems. For this paper we have shown the evaluation of two off-the-shelf products for business messaging, RabbitMQ which is an implementation of AMQP protocol vs. Java-WebSockets which uses WebSocket protocol [RFC6202], keeping the same rule system (Drools Expert) for both.

Figure 4 shows the comparison results for products RabbitMQ (left) and Java-WebSockets (right). It shows four different time graphs, namely, Total Time (A), Rules Execution Time (B), Wait Time (C) and Publish Time (D) for both RabbitMQ and Java-WebSockets.

Total time comparison (Figure 4-A) shows that Java-WebSockets perform much better than RabbitMQ in terms of overall system performance as per our requirement’s setup.
In the start upto 10k events both perform well but for more than 10k events RabbitMQ’s performance decreases and it takes way too much time for 50k events. However Java-WebSockets handles the high data stream of 50k easily and also can be scaled for further more events with ease.

Figure 4-B shows the Rules execution time, in other words it shows the rule engine’s (Drools Expert) performance. An interesting point to note here is the rule execution time in CEP for Java-WebSockets has unexpectedly increased. Reason behind this is out of scope of this paper and will be investigated as a future extension of the work. For now, we just assume that due to faster messaging, load on rule engine increases and hence affects the processing.

There is a wait state at each node, since each cascaded node waits for the event to be received and then processes it. Figure 4-C compares the wait times of all nodes between RabbitMQ and Java-WebSockets. Java-WebSockets messaging is very fast and all the nodes have negligible or acceptable wait time, but for RabbitMQ, CEP wait time increases drastically after 10k events. This is the wait CEP does for consuming events from the queue. Also from Figure 4-D, the publish time of various nodes, we could note that input node publish time in RabbitMQ has also increased too much after 10k. Both the nodes, INPUT node and the CEP node are always accessing the same queue simultaneously; for INPUT to publish on the queue and for CEP to consume from it.

At high event stream (above 10k), the queue gets a lot of data buffered inside and CEP takes longer time due to slow parsing involved for converting data from bytes to Maps whereas, the conversion from Strings to Maps, which occurs in WebSockets is considerably faster. This wait time accounts for overall degraded performance our system while using RabbitMQ for messaging.

With these results we conclude that, RabbitMQ performance is good for low number of events and low volume of data, as the number of events increases above 10k, queues get lots of data buffered which substantially slows down the overall system. For our system architecture, Java-WebSockets are found to be a favorable replacement over RabbitMQ and it makes the overall system more efficient and highly scalable.

VII. SUMMARY AND FUTURE WORK

This paper describes the third phase of our work on building an architecture for rule-based event processing distributed system, which combines a messaging system with a rules system. We start by describing the underlying technologies, tools and products being used. Messaging using AMQP is implemented by RabbitMQ and using WebSockets is implemented by Java-WebSockets. Our rule system used common to both the messaging systems is Drools-Expert.

The architecture we have created consists of several interconnected components with communication links, realizing a distributed system. In our architecture we introduce 2 rule governing nodes; Complex Event Processing (CEP) and Distributed Governance (DG). We have streams of events entering the system which are being processed by CEP to generate complex events, essentially identifying patterns within the events. These complex events are then fed into DG for analysis and decisive action. The communication links between these components is provided by the messaging system.

In our previous paper, we focused on the evaluation of performance of the RabbitMQ alone with different data-types in java (Strings vs. Maps) by running several tests with events ranging from 10 to 100 thousand. The wait state, introduced due to dependency of a node on processing time of previous node, is also considered.

In this paper we extended our work by comparing two off-the-shelf business messaging products; RabbitMQ vs. Java-WebSockets. We critically analyzed the performance of the system with WebSockets and compared the results with our previous work for RabbitMQ.

Part of the future work planned is to deploy multiple CEP and DG nodes/engines on multiple machines that can work simultaneously to distribute the load at required times. We also have planned to increase the complexity of the governing rules in CEP and DG to test the highly complex patterns matching. A higher performance is the main objective of our work, currently we have all our nodes tested under constrained environment, working on a single machine (Intel i5, dual core) with Windows 7. Running our nodes across distributed servers in a cloud-based deployment should see the approach scale to a level appropriate for a high throughput, telecommunication grade management process.

REFERENCES

RabbitMQ (AMQP) | Java-WebSockets (WebSocket)
---|---
![Graph A](image1.png) | ![Graph A](image2.png)
![Graph B](image1.png) | ![Graph B](image2.png)
![Graph C](image1.png) | ![Graph C](image2.png)
![Graph D](image1.png) | ![Graph D](image2.png)

Figure 4. Example of a TWO-COLUMN figure caption: (a) this is the format for referencing parts of a figure.