Cloudifying Mobile Network Management: Performance Tests of Event Distribution and Rule Processing

Sumit Dawar[1,2], Sven van der Meer[1], John Keeney[1],
Enda Fallon[2], and Tom Bennet[2]

[1] Ericsson Network Management Labs,
Ericsson Software Campus, Athlone, Co. Westmeath, Ireland
{sumit.dawar,sven.van.der.meer,john.keeney}@ericsson.com
[2] Athlone Institute of Technology, AIT, Athlone, Co. Westmeath, Ireland

Abstract. With the ever increasing number of devices, nodes and the events they create, scalability and performance become important aspects for Operation Support Systems (OSS). One solution is to distribute the work load, i.e. 'cloudify' the formerly centralized monitoring and decision functions. This requires remodeling Complex Event Processing (monitoring) and Policies (decision making) towards a distributed yet coordinated system. This paper describes an extended architecture, implementation and performance tests for a policy-based event processing system. The main advantage of our approach is that we use policies for event pattern matching (an advanced form of Complex Event Processing) and for the selection of corrective actions (called Distributed Governance). Policies are (a) distributed (over multiple components) and (b) coordinated (using centralized authoring). The resulting system can deal with large numbers of incoming events, as is required in a telecommunication environment. Peak load will be well above 1 million events per second, combining different data sources of a mobile network. This paper presents the motivation for such a system, along with a comprehensive presentation of its design, implementation and evaluation.

Keywords: Complex Event Processing, Rule System, Distributed Processing, Performance.

1 Introduction

Mobile networks are growing, cell sizes are decreasing and the number of connected devices is exploding. These conditions result in an ever increasing number of events from the network. The situation becomes critical and requires scalable solutions for event processing and the selection of corrective actions, i.e. for alarm events. 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 [5-8], which drastically limits the scalability. We use distributed rules that are coordinated by centralized authoring to address this limitation. In this paper, we describe the general architecture, the reference implementation we have

D. Pesch et al. (Eds.): MONAMI 2013, LNICST 125, pp. 94–107, 2013.

developed and performance tests with regard to end-to-end message processing. The work is integrated into a wider research project in the Ericsson Network Management labs that deals with extreme volumes of mobile network events.

This paper is organized as follows: section 2 introduces core concepts, technologies and products from messaging systems and rule systems. Section 3 briefly discusses the architecture and main design decisions of our system. The sections 4 and 5 then detail the implementation and provide a discussion of test results, mainly looking into the performance for the end-to-end event processing. Section 6 discusses related work from academia and industry. Finally, a conclusion summaries this paper and discusses future work items of our project.

2 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.

2.1 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 works use Advanced Message Queuing Protocol (AMQP) messaging due to external project requirements, namely RabbitMQ, an open source AMQP implementation.

AMQP is “an open standard for passing business messages between applications” [1]. Data (the messages) is sent in a stream of octets, 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.

2.2 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). [4]

3 Architecture and Design

We receive events from streams (using other Ericsson software), 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 (CEP) 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 un-decidable 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.

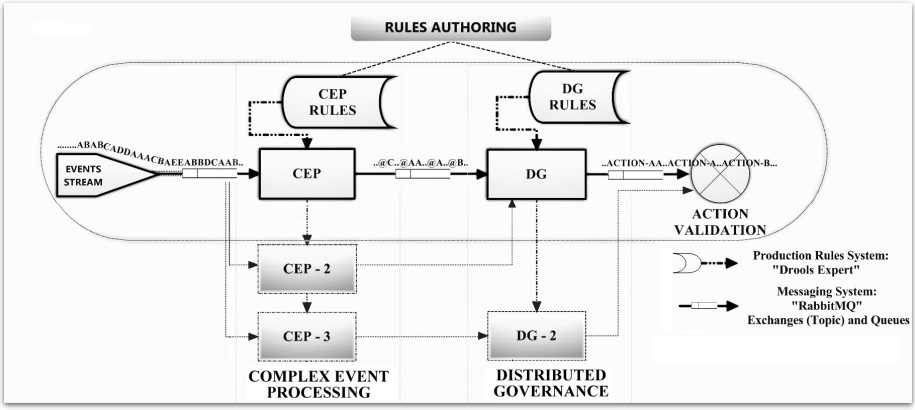

Fig. 1. Architecture and deployment scenario

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.

4 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).

4.1 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.

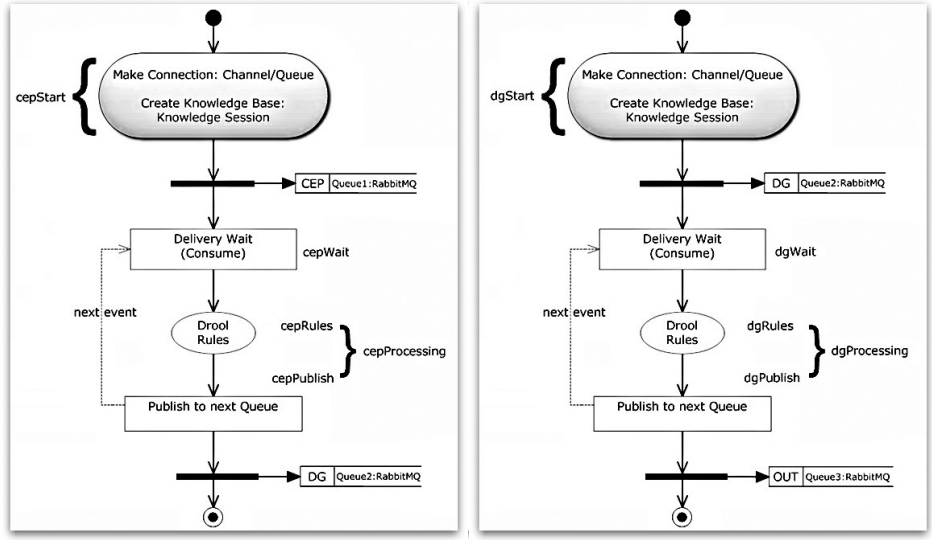

Fig. 2. Core Nodes, CEP (left) and DG (right)

4.2 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.

4.3 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 1. Single and Multi-event Pattern (examples)

Single event Pattern		
Incoming Event	Composite Event	Associative Action
A	@A	Action-A
B	@B	Action-B
C	@C	Action-C
D	@D	Action-D
E	@E	Action-E

Multi-event Pattern		
Incoming Events	Composite Event	Associative Action
A-A	@AA	Action- AA
A-B†	@AB	Action- AB
A-A-B	@AAB	Action- AAB
A-A-B-B	@AABB	Action- AABB

†A-B implies that 'B' occurs after 'A'

Table 1 shows an example of events (single and multi) and corresponding complex events with associative action. This pattern matching can be extended to generate new complex events, by simply writing the new CEP rules and corresponding rules in the DG for associative action.

4.4 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.

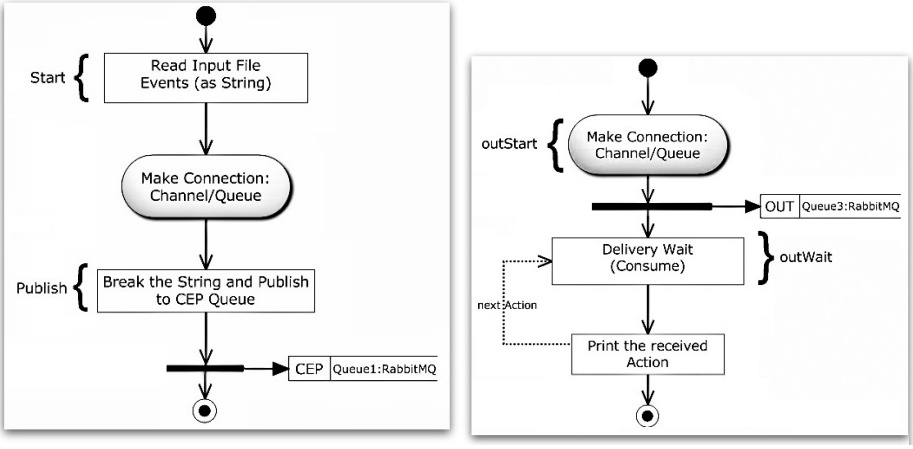

Fig. 3. Input (left) and Output (right) Console support Nodes

5 Testing and Evaluation ("Strings" as Alarm Events)

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 10 up to 1,000,000 events in a single stream with 10 test runs per input stream size. 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 a 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)

Figure 4 shows the overall processing time, i.e. the time it takes to process all events from input console to output console. The time for up to 1,000 events is negligibly small. From 10,000 events onwards the time rises in proportion with the increasing number of event in the input stream.

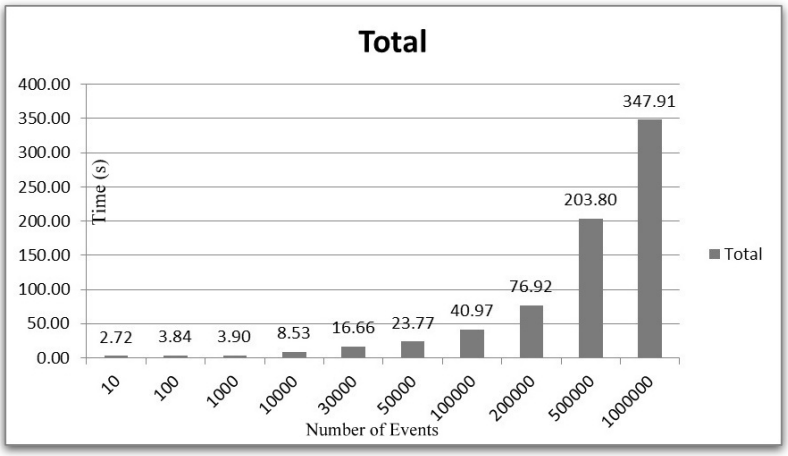

Fig. 4. Averaged Maximum Time for system (with String events)

5.1 Time Consumption on Core Nodes

The different times within CEP and DG namely Start, Wait, Rules, Publish, Processing and Total are measured and plotted on the graphs shown in Figure 5. The initialization phase of the nodes (Start) has been included here to show that it has no impact on the overall system performance (note: the number of rules and topics did not change).

An important metric evaluated is the time consumed during rule evaluation and the wait a node does before fetching the next event from the queue. These times, Wait and Rules, shown in Figure 6 and discussed in the following section, depict the performance of Drools Expert.

Another Important metric is the time each node takes to publish events to the topic. There are three nodes doing this task on their corresponding topics; the input console, CEP and DG. The publish time of these nodes measures the efficiency of RabbitMQ.

5.2 Discussion

Drools Expert rules are used on the nodes CEP and DG. DG's Wait is directly proportional to the time the CEP node takes for rule evaluation. The output console also waits with DG for CEP rules evaluation and then waits for the DG rules evaluation. Hence it has the longest wait time. Figure 6 compares wait time with the rules evaluation time with increasing number of events.

CEP rules are complex and identify patterns that take more time compared to DG rules which are used to select actions associated to patterns.

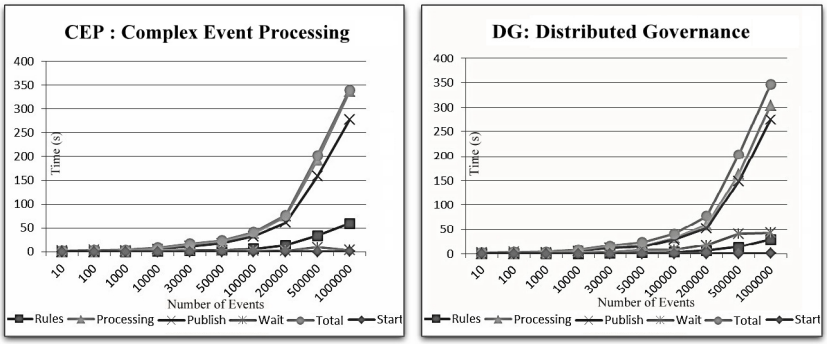

Fig. 5. Different time consumptions in CEP and DG

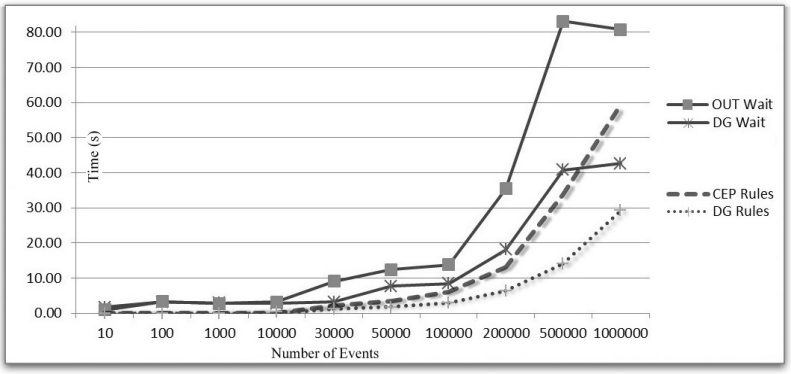

Fig. 6. 'DG and OUT node Wait' vs 'CEP and DG Rules'

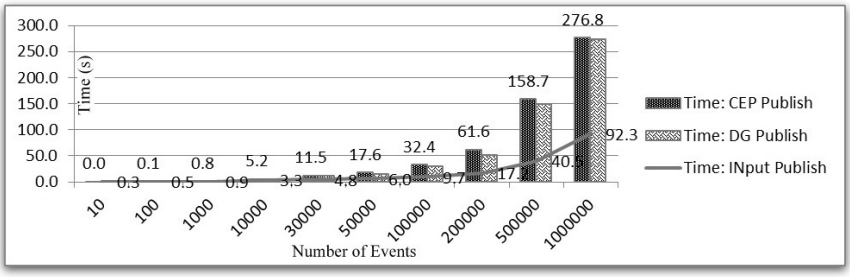

Fig. 7. Publish time for Input, CEP and DG nodes

There are three locations where events are published: CEP, DG and the input console. Figure 7 shows the time it takes to publish. The time taken by the input console to publish all the events is very small (virtually negligible) for up to 30,000 events finishing even before CEP starts processing events.

Above 30,000 events, as the number of events increases it affects the CEP processing time and generates a cascading effect for the overall system performance. Thus, separating the input console from CEP (and subsequently DG) is important for any event stream above 30,000 events.

6 Testing and Evaluation ("Maps" as Alarm Events)

Extending our distributed system to work with Maps (*LinkedHashMap<String, Object>*). Maps are closer to the real world mobile network alarms and by hence replacing Strings with Maps (alarms); we will be able to find out the real world performance of our distributed system. The main information that a Map event contains are the alarm ID, the language, its type, timestamps and payload. With Maps we have a flexibility to add some critical information during processing that could help in final decision for action in response to the alarm.

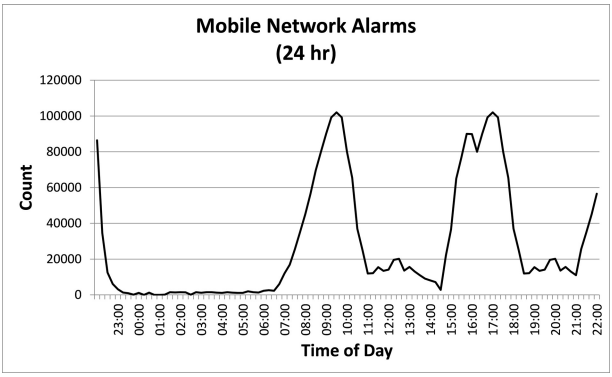

Fig. 8. Number of Mobile Network Alarms during a day

Figure 8 shows a possible event pattern of a mobile network for 24 hour period. During the event storms (at the peaks in figure above), we require a system that could work seamlessly and provides for an efficient alarm management system.

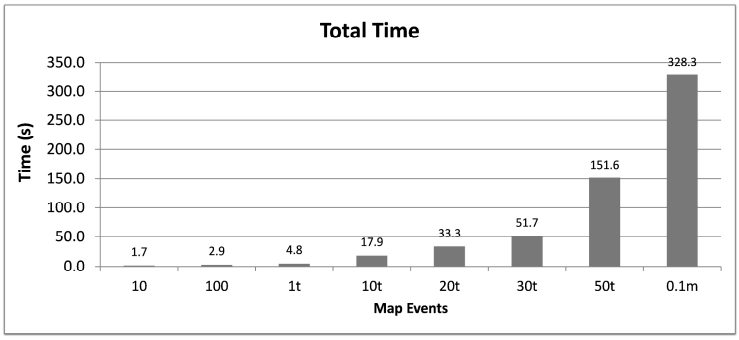

Fig. 9. Averaged Maximum Time for System (with Map events)

We need our system to perform at a speed of thousands of events per second. So that with multiple nodes we could make it more efficient for congestion free peak hours. For such performance the response time of a system should not be more

than 5-6 minutes for an event storm. We have kept our experiments bound to a limit of 5 minutes of total system run. We went up to 1 million events with Strings but with Maps we are able to go up to 100k events within our time bound.

Figure 9 shows the mean processing time for Map events, for 100k events it exceeds 5 minutes which is considered under our test environment but not acceptable in real world mobile networks scenario.

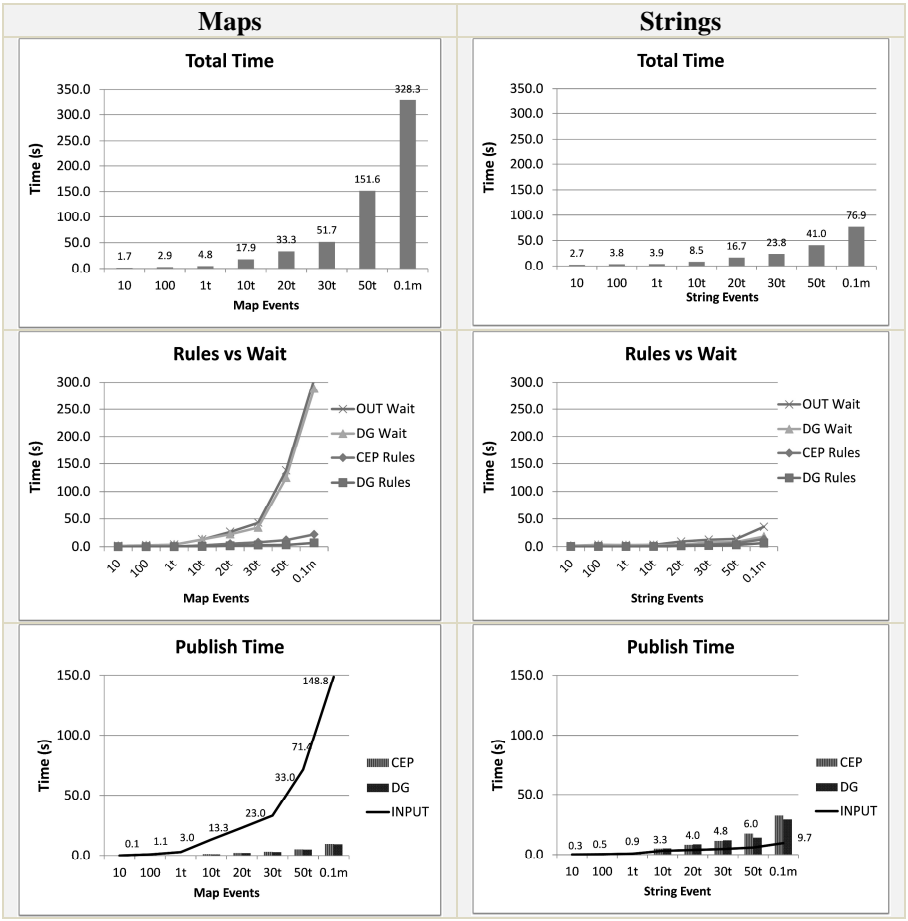

Fig. 10. Map Events vs String Events

6.1 Discussion: Map Events vs String Events

The comparison of Maps as events to the Strings as events is shown in Figure 10. The important difference we noted by the comparison is that input takes a very long time to publish as the number of events increase. This increases the wait time of cascaded system (CEP-DG-OUT) and hence the total processing time. For Maps

messaging system has to hold and process more than 10 times of data per event compared to Strings and hence becomes slow.

We observe that RabbitMQ is not the best for messaging system when the load becomes high during events storm and hence we get a backlog processing at that time; which is not be acceptable for real world mobile networks. We have considered a couple of alternatives for messaging system which are under testing to be made compatible and function in parallel with our rule processing system.

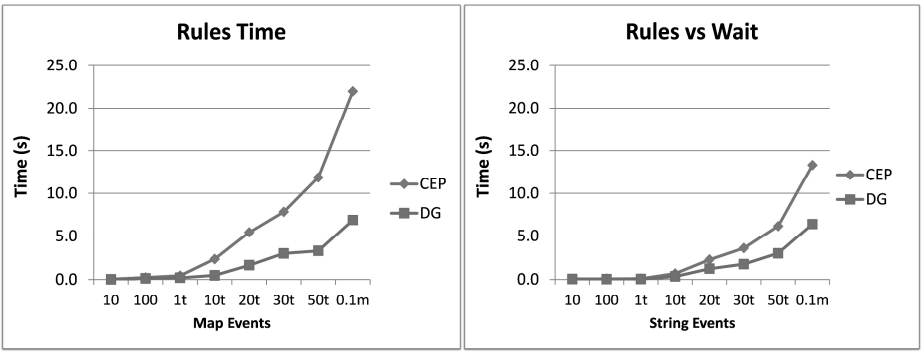

Fig. 11. Rules Processing time: Maps and Strings

Rules for CEP and DG are re-written to work with Maps but rules processing. For CEP which holds the knowledge of 4 concurrent events for pattern matching, it takes longer now with heavier data of maps. But for DG it is almost equal as the rules are simple and no pattern matching is done.

This time consumption is not critical as it is still under 25 seconds for 100k events. With highly complex pattern matching, which is a part of our future work, we can safely assume that rule system would take longer time without going critical.

We conclude that with maps we have gathered a better analytics for the performance of our distributed system in real environment.

7 Related Work

In the Policy-Based Information Sharing in Publish/Subscribe Middleware [5] 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 [6] 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 [7] 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 [8] 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 rules 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.

8 Summary and Future Work

This paper describes the second phase of our work on building a 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 our rule system uses 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. There are topic exchanges (channels and queues) between components which provide forwarding with selective filtering capability.

In our previous paper, we focused on the evaluation of performance of the products RabbitMQ and Drools by running several tests with events ranging from 10 to 1 million. In effect we are measuring the performance of rules and publishing of complex events. 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 for evaluating the performance and take to closer to the real world alarm events. We introduced Maps as events with several parameters stored inside one event. Then we critically analyzed the performance of the system and compared the results with our previous work with Strings as events.

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

1. AMQP Architecture, http://www.amqp.org/architecture (last visited: February 22, 2013)
2. RabbitMQ Tutorial, http://www.rabbitmq.com/tutorials/tutorial-one-java.html (last visited: February 22, 2013)
3. RabbitMQ AMQP Concepts, http://www.rabbitmq.com/tutorials/amqp-concepts.html (last visited: February 22, 2013)
4. JBoss.org, http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/pdf/drools-expert-docs.pdf (last visited: February 1, 2013)
5. Singh, J., Vargas, L., Bacon, J., Moody, K.: Policy-Based Information Sharing in Publish/Subscribe Middleware. In: IEEE Workshop on Policies for Distributed Systems and Networks, POLICY 2008, Computer Lab., Univ. of Cambridge, Cambridge, June 2-4, pp. 137–144 (2008)
6. Paschke, A., Kozlenkov, A.: A rule-based middleware for business process execution. In: Multi-konferenz Wirtschaftsinformatik, MKWI (2008)
7. Erradi, A., Maheshwari, P., Tosic, V.: Policy-Driven Middleware for Self-adaptation of Web Services Compositions. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 62–80. Springer, Heidelberg (2006)
8. Winn, G.M., Young, N.G.S.: Message oriented middleware with integrated rules engine International Business Machines Corporation: US Patents US20130007184 (2012), http://www.google.com/patents/US20130007184