Mete: a Meta Rete Interface for Distributed Rule-based Systems

Maarten Hubrechts[*], Kennedy Kambona[†], Thierry Renaux[‡], Simon Van de Water[§], Mathijs Saey[¶], Coen De Roover[‖] and Wolfgang De Meuter[**]

Software Languages Lab, Vrije Universiteit Brussel
Brussels, Belgium

Email: [*]maarten.hubrechts@vub.be, [†]kennedy.kambona@vub.be, [‡]thierry.renaux@vub.be, [§]simon.van.de.water@vub.be, [¶]mathijs.saey@vub.be, [‖]coen.de.roover@vub.be, [**]wolfgang.de.meuter@vub.be

***Abstract*—In Complex Event Processing (CEP), continuous streams of information are analyzed to deduce useful insights from the incoming data. CEP can be implemented using rule-based systems. These are systems which are used to act upon states in which an object domain can be. It does so by providing facts that represent that object domain and rules that describe when certain actions must happen. In order to determine this, a rule interpreter is used to find the rules that match the available facts. A widely used example of such an interpreter is the Rete algorithm. However, since Rete was first introduced in the seventies, changes to the algorithm are needed to make it relevant to today's standards in CEP. One of those changes is the support for deployment in a cloud setting or on a computer cluster to enhance the processing performance. Previous research describes a distributed variant of the Rete algorithm which distributes the workload across a computer cluster. Inescapably, distributing the work over different machines introduces additional meta-concerns such as load balancing and fault tolerance.**

In this paper, we claim that the complexity of maintaining CEP systems can be reduced by disentangling these meta-concerns from the base application. We introduce a meta Rete system called *Mete*. Mete reasons about, and intercedes in, a distributed base Rete network. Mete allows programmers to add and modify meta-concerns such as auditing, fault tolerance, load balancing, or logging, without requiring any modification to the base application code.

***Index Terms*—CEP, Meta Programming, Rete, Rule-based Systems**

I. Introduction

In today's world enterprises continuously generate a substantial amount of data. This leads to never ending streams of information. Yet, the true potential of having this data is only reached when it is analysed; processing data allows enterprises to gain useful insights. This is where *Complex Event Processing*, or *CEP*, comes into play. CEP is a method of analyzing continuous streams of data and deriving conclusions based on them [1]. The data itself concerns actions that happen in the scope of the enterprise. These data points are called *events*. A combination of events is called a *complex event*. Complex events are typically some situation, opportunity or threat to the enterprise that must be acted upon. Hence, when complex event gets detected, a certain action is executed as a consequence.

One way of implementing Complex Event Processing is by using a *rule-based* approach [2]. In rule-based CEP systems, declarative rules are used to identify complex events. Declarative rules are combinations of a pattern and a consequent; when input data matches the pattern, the consequent should be executed. In order to determine which rules are matched by the available facts, a *rule interpreter* is needed [3].

A widely used example of such an interpreter is the Rete algorithm [4]. Rete is used to efficiently match many facts to many patterns. However, the traditional algorithm was developed for single-threaded use. This means that the algorithm does not specify how to take advantage of the additional compute power of multi-core computers or computer clusters. This problem was tackled by [5] who adapted the algorithm to operate on a machine consisting of a large number of processing elements, each with its own local storage. Another extension based on this by [6], introduced the distributed rule-based CEP system CloudParte. Such distributed variants of Rete made it possible to scale up rule processing by distributing the work over different machines.

However, a distributed variant of the Rete algorithm comes with a new set of challenges. Examples of such challenges are making the algorithm resistant to partial failures of the computer cluster and implementing a load balancing technique to evenly distribute the work over the different machines. These challenges are all meta-concerns. Ideally, implementing and modifying solutions to these challenges should be done in such a way that the application logic is left unaffected. By disentangling the concerns, adaptive and perfective maintenance [7] could be significantly simplified.

So far, there is no abstract and general way to tackle the meta-concerns of a distributed rule-based CEP system. While the declarative nature of rule-based systems naturally lends itself to disentangling the meta-concerns from the base application logic, current rule based systems impose a hard line between the declarative rules and their implementation. While changes to the application logic require rewriting the declarative rules, user demands for enhancements and extensions to the rule-based system require rewriting elements of the rule-based system's core implementation. In practice this entails that modifications are implemented by writing software in the rule-based system's implementation language. For far-reaching concerns such as distribution, fault tolerance, and load balancing, this requires the programmer to fully understand the entire code base of the rule-based system.

We propose to instead tackle such meta-concerns using a meta Rete protocol. Through such a meta Rete protocol, programmers can introspect the workings of a rule-based system, and intercede in its workings. Crucially, the meta Rete protocol is exposed to the rule-based language as logical facts and actions which can be invoked from the consequent of a rule matching.

We developed a prototypical meta Rete system called *Mete*. Mete provides an abstract and general way to make extensions to a distributed CEP system based on the Rete algorithm. The novelty of Mete is that it introduces a second Rete network, called the *meta Rete network*. The meta Rete network reasons about the *distributed base Rete network*. This enables us to write meta rules which can make fundamental changes to the base Rete network in a declarative manner, which reduces the burden of maintenance and configuration. Tackling other meta-concerns, or modifying the existing solutions, can be done without having to touch — nor know the details of — either the base application logic or the rule-based system's implementation.

Section II describes the architecture of Mete. In section IV we sketch an evaluation of our prototype. In order to validate our system, we implement the meta-concern of fault tolerance using only declarative meta-rules. To verify that the system is indeed fault tolerant, we run it with and without induced failures, and confirm that the results computed by the base Rete are unaffected by partial failures of the computer cluster it runs on.

II. Architecture

The Rete algorithm was initially designed for single-threaded use. However, these days we are dealing with data on too large a scale too tackle that way. Much more processing power can be brought to bear by running workloads on multiple cores or even multiple machines. This is why variants of Rete were introduced that can be parallelized [8]–[12] or distributed across a computer cluster [5], [6]. All those systems must take care of the additional challenges that present themselves in a parallel or distributed context. For instance, the algorithm must be made resource-aware. For distributed cases, the algorithm must be made resistant against partial failures of the hardware.

Our work expands on the distributed Rete system CloudParte [6]. CloudParte splits up a Rete network and spreads the Rete nodes among multiple machines that are part of the computer cluster CloudParte runs on. CloudParte provides the means for asynchronously exchanging messages between the Rete nodes. However, CloudParte's solution to the challenges of a distributed setting are hard-coded. Extending its capabilities, or changing the trade-offs that were made, requires modifications to its core implementation. This is cumbersome and error-prone, and discourages maintenance and adaptation.

In order to facilitate evolving CloudParte's code base, we introduce a meta Rete system called *Mete*. Mete extends a CloudParte system by providing a secondary Rete network. In this secondary Rete network, meta-concerns of the *base* Rete network are reified as logical facts. Logical rules in this secondary Rete network hence reasons about the base Rete network, effectively constituting a *meta Rete network*. We refer to the rules of the meta Rete network as *meta rules*. The facts in the meta Rete network are called *meta facts*.

More concretely, the meta facts describe how the distributed base Rete network is structured. For example, they define which nodes are present in the base system and on which machine they are located. The meta Rete network is responsible for spawning the base Rete network, restoring failed nodes, and relocating nodes to balance the load.

When Mete initializes, it first spawns the meta Rete network on a single machine, called the *master machine*. Then, the distributed base Rete program is parsed and compiled into meta facts representing it. Next, the meta facts are passed to the meta Rete network. Since the initial situation is merely an exceptional case of a valid configuration — all nodes of the base Rete are present but none are scheduled on a machine — the meta Rete can perform its function by spawning the base Rete network on the different machines.

Not all meta rules are devoted to spawning the base Rete network. Some meta rules can, e.g., implement checkpointing strategies. In general, meta rules provide solutions to *any* user requirement that might present themselves in the future.

Summarized, the system contains two active Rete networks, shown in fig. 1:

1) The first network is the meta Rete network which has the distributed base Rete network as its object domain. Meta rules allow to describe situations that the distributed base Rete network can be in. These rules make it possible to influence and extend the behavior of that base network. This means that the base Rete network's behavior can be adapted according to new requirements, which means that the algorithm can evolve without knowing a priori what these requirements are.
2) The second network is the base Rete network. The base Rete network reasons about a specific application domain, e.g., a supermarket's customers and their purchases. The base Rete network can be distributed across multiple physical machines, in which case we refer to it as the distributed base Rete network.

III. A Meta Rete Interface

The meta Rete network implements a specific *meta Rete interface*. This interface specifies which constructs are available to *user-defined meta rules*. User-defined meta rules provide a way for application developers to extend and configure the CEP system. They reason about *meta facts*, i.e., facts that represent the state and structure of the distributed base Rete network.

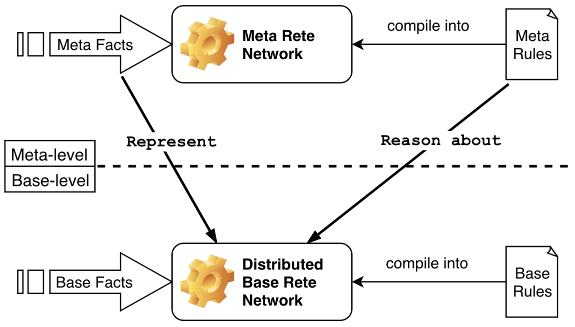

Fig. 1. Schematic representation of the architecture of Mete.

The meta interface consists of two parts. First, there are the meta facts produced by the rule engine. These can be matched as logical facts by the user-defined meta rules. Second, there are the meta facts that are understood by the rule engine. These can be asserted into — or retracted from — the rule engine's knowledge base by user-defined meta rules. The former enables the meta-rules to respond to the state of the rule engine. The latter enables meta-rules to modify the state of the rule engine.

The meta facts by which the state of the rule engine can be inspected, are shown in Listing 1. They are divided in five different categories:

Architecture patterns match meta facts which represent the structure of the distributed base Rete network. For example, line 2 shows a pattern of a *Node* meta fact. This pattern matches every Rete node that is part of the distributed Rete network, since each such node is represented in the knowledge base by a *Node* meta fact. Another example is the pattern describing *Edge* meta facts, shown on lines 7 and 8. This pattern matches the meta facts of all edges that connects two Rete nodes of the distributed base Rete network.

Timing patterns match *TimerElapsed* meta facts. The presence of a *TimerElapsed* meta fact in the knowledge base indicates that a certain timers threshold has elapsed.

Info patterns match meta facts containing additional information about a base Rete node that is not related to its structure. A concrete example are *LoadCheck* meta facts. These communicate the amount of facts that the corresponding base Rete node forwarded to its successor(s). Info meta facts can for example be useful for implementing a load-balancing strategy.

Snapshotting patterns match meta facts that are related to snapshotting [13] the state of the different Rete networks.

Wildcard patterns are all patterns that are automatically matched exactly once, when the system gets started. Wildcard patterns can be used to initialize components of the meta Rete network, or to initialize the distributed base Rete network when the system gets started.

```
1  Architecture
2  [Node (id: NodeId)]
3  [AlphaNode (id: NodeId)]
4  [BetaNode (id: NodeId)]
5  [NodeF (node_id: NodeID, f: Fct]
6  [NodeInputs (node_id: NodeId, sides: Sides)]]
7  [Edge (from_id: FromNodeId, to_id: ToNodeId,
8         side: Side)]
9  [NodePid (node_id: NodeID, node_pid: NodePid)]
10 [ManagerPid (manager_id: ManagerId,
11              manager_pid: ManagerPid)]
12 [Location (node_id: NodeId, manager_id: ManagerID)]
13
14 Timing
15 [TimerElapsed (timer_id: TimerId)]
16
17 Info
18 [LoadCheck (id: NodeId, output_log_size: OutputLogSize)]
19
20 Snapshotting
21 [Snapshot (id: NodeId, timestamp: Timestamp,
22            sides: Sides, output_log: OutputLog)]
23 [AgendaSnapshot (timestamp: Timestamp,
24                  activations: Activations)]
25
26 Wildcard
27 [_ ()]
```

Listing 1. Patterns of meta facts that can be matched in the patterns of user-defined meta rules.

The meta facts by which the rule engine's state can be modified, are shown in Listing 2. To modify the state, meta facts matching these patterns can be asserted into the rule engine's knowledge base — or retracted from the knowledge base. There are three categories of meta facts by which the rule engine's state can be modified:

Timing meta facts make or remove timers. When a *RequestTimer* meta fact gets asserted a timer is created. Whenever the specified time is up, a *TimerElapsed* meta fact (described previously) is asserted into the knowledge base.

Info meta facts serve a similar role, but instead of requesting timer events, they request state-introspection meta facts from the Info category.

Snapshotting meta facts request different kinds of snapshots to be made. Line 8 shows the meta fact *RequestSnapshot* that when asserted, fires a built-in meta rule that will assert a *Snapshot* meta fact (described previously). Line 9 shows the *RequestMetaSnapshot* that when asserted, fires built-in meta rules that take a snapshot of the meta Rete network itself, and stores it to persistent storage.

```
1 Timing
2 [RequestTimer (timer_id: TimerId, ms: Ms)]
3
4 Info
5 [RequestLoadCheck (node_id: NodeId)]
6
7 Snapshotting
8 [RequestSnapshot (node_id: NodeId)]
9 [RequestMetaSnapshot ()]
```

Listing 2. Patterns of meta facts that can be asserted in the consequents of user-defined meta rules.

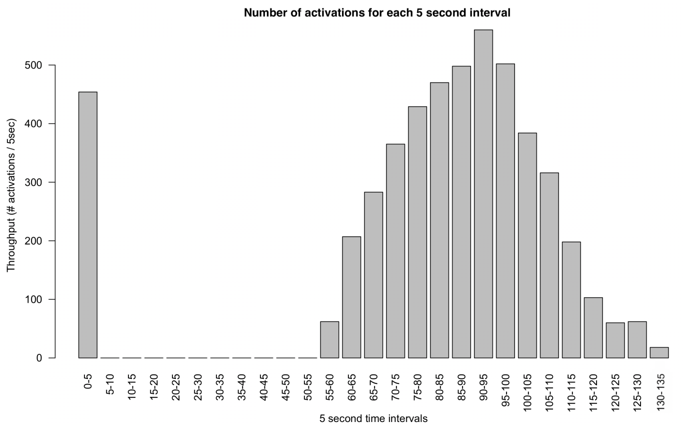

Fig. 2. Throughput (in activations per 5 second interval) with a simulated failure of entire distributed Rete network.

IV. Evalution

In order to validate *Mete*, we implemented fault tolerance as a set of user-defined rules, using only the meta Rete API from section III. Together, these meta rules ensure that any node involved in either the distributed base Rete network or the meta Rete network can fail without influencing what the rule engine detects. For space constraints and to limit the scope of this paper — we introduce a meta Rete interface, not fault tolerance — we do not list the exact set of meta rules by which we implemented fault tolerance. The gist is as follows: each situation in which a fault of a component would lead to a global failure is handled by one or more user-defined rules. These make sure that each node (*Node*) is regularly (*RequestTimer*/*TimerElapsed*) persisted (*Request-Snapshot*/*RequestMetaSnapshot*), and respawn the nodes with that state after a crash.

We conducted an experiment where 5000 facts were sent to the base Rete network, which contains a rule which fires whenever one of those facts arrives. That base rule hence fires exactly 5000 times. After a delay of five seconds, we induce a failure in all the nodes of the distributed base Rete network. As fig. 2 shows, the user-defined meta rules succeed in respawning the base Rete nodes. After the induced failure, no base rule fires for a period of fifty seconds. After this period, the rule gets fired again until all 5000 activations took place. This demonstrates that our system is robust against crashes of the distributed base Rete network.

V. Conclusion

In this paper we propose an abstract and general way of extending the capabilities of rule-based systems running on the Rete algorithm in a distributed setting. Previously, extending the algorithm had to be done by manually adding language-specific code directly into the implementation. However, this approach is undesirable because it lacks the advantages that stem from suitable software maintenance techniques such as perfective maintenance.

We introduce Mete, a meta Rete system that reasons about a base Rete network by means of a set of special rules and facts. The architecture of Mete contains two active Rete networks. The distributed base Rete network is reasons about the object domain of the application via facts and user-defined rules. Mete then provides a meta Rete network — a non-distributed network that reasons about the base Rete network using meta facts and meta rules. The distributed base Rete network is represented by meta facts, describing its state and structure. Built-in meta rules are provided that can match these meta facts. The meta rules describe the extensions to the behavior of the distributed base Rete network. We validate our approach by enforcing fault tolerance in a distributed rule-based system. By introducing Mete, perfective maintenance is improved since it offers an abstract and general way of extending the Rete algorithm.

References

[1] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] Adrian Paschke and Alexander Kozlenkov. Rule-based event processing and reaction rules. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 53–66. Springer, 2009.

[3] Frederick Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, 1985.

[4] Charles L. Forgy. Expert systems. chapter Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[5] Michael A Kelly and Rudolph E Seviora. A multiprocessor architecture for production system matching. In *AAAI*, pages 36–41, 1987.

[6] Janwillem Swalens, Thierry Renaux, Lode Hoste, Stefan Marr, and Wolfgang De Meuter. Cloud parte: elastic complex event processing based on mobile actors. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*, pages 3–12. ACM, 2013.

[7] Bennet P Lientz and E Burton Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981.

[8] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel algorithms and architectures for rule-based systems. *SIGARCH Comput. Archit. News*, 14(2):28–37, May 1986.

[9] Mostafa M. Aref and Mohammed A. Tayyib. Lana-match algorithm: A parallel version of the rete-match algorithm. *Parallel Computing*, 24:763–775, 1998.

[10] Kemal Oflazer. Partitioning in parallel processing of production systems. 1987.

[11] A Gupta, CL Forgy, D Kalp, A Newell, and M Tambe. Parallel ops5 on the encore multimax. In *Proc. Int'l Conf. on Parallel Processing*, pages 271–280, 1988.

[12] Toru Ishida. Parallel rule firing in production systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):11–17, 1991.

[13] E. Gendelman, L. F. Bic, and M. B. Dillencourt. An application-transparent, platform-independent approach to rollback-recovery for mobile agent systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 564–571, April 2000.