Cloud PARTE: Elastic Complex Event Processing based on Mobile Actors

Janwillem Swalens Thierry Renaux Lode Hoste Stefan Marr Wolfgang De Meuter

Software Languages Lab
Vrije Universiteit Brussel, Belgium
{jswalens,trenaux,lhoste,smarr,wdmeuter}@vub.ac.be

Abstract

Traffic monitoring or crowd management systems produce large amounts of data in the form of events that need to be processed to detect relevant incidents. Rule-based pattern recognition is a promising approach for these applications, however, increasing amounts of data as well as large and complex rule sets demand for more and more processing power and memory. In order to scale such applications, a rule-based pattern detection system needs to be distributable over multiple machines. Today's approaches are however focused on static distribution of rules or do not support reasoning over the full set of events.

We propose Cloud PARTE, a complex event detection system that implements the Rete algorithm on top of mobile actors. These actors can migrate between machines to respond to changes in the work load distribution. Cloud PARTE is an extension of PARTE and offers the first rule engine specifically tailored for continuous complex event detection that is able to benefit from elastic systems as provided by cloud computing platforms. It supports fully automatic load balancing and supports online rules with access to the entire event pool.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent programming; D.3.4 [*Programming Techniques*]: Processors; I.5.5 [*Pattern Recognition*]: Implementation

General Terms Algorithms, Design, Performance

Keywords mobile actors, Rete, complex event processing, online reasoning, load balancing

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

Large-scale processing of complex events gains new relevance with the *big data* movement and an increasing availability of real-time data, for instance for enterprise applications, or societal and security-related scenarios. Companies want to leverage real-time big data for new products and improved services [1], while traffic and crowd monitoring systems could be used to prevent traffic jams or guide rescue services in emergency situations.

Such applications require flexible and maintainable event processing systems that can be adapted easily for instance to recognize new emergency scenarios or business cases. One approach is to use machine learning techniques to analyze, filter, and categorize events. However, these approaches are often black boxes that do not give feedback on the reasons for a certain classification [12, 13]. Moreover, most machine learning techniques require training data, which is seldomly available for exceptional and emergency situations.

For scenarios where training data is lacking or hard to gather, query and rule-based complex event detection (CED) systems are more suitable choices [11]. Furthermore, rule-based systems have the advantage of providing software engineering abstractions to express intent, avoiding cumbersome and error-prone ad-hoc implementations, and allowing expert programmers to intervene in the correlation of events by explicitly encoding application logic. For instance, the forming of a crowd can be detected based on simple rules that correlate location and movement of people in public places. Similarly, upcoming traffic jams can be detected with rules that detect deviations from the typical speed of cars.

While rule-based systems are appealing, the applications listed above need to process large amounts of data efficiently in order to reach the responsiveness required for use cases that are based on live feedback. One application that could be broadly described as "crowd monitoring" is the real-time analysis of a soccer game, as proposed by the ACM DEBS 2013 Grand Challenge.[1] While most participants of

[1] *The ACM DEBS 2013 Grand Challenge*, DEBS, access date: August 15, 2013 `http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails`

this competition prototyped their systems with rule-based complex event detection systems such as Esper,[2] the final systems used highly custom solutions, severely lacking software engineering abstractions and maintainability, and with strong limitations on the rules. Today's rule-based systems do not yet provide the necessary efficiency to handle such use cases with the required performance.

In order to facilitate big data applications that are supposed to provide live feedback, complex event detection engines and pattern recognition systems need to be able to deal with data sets that are larger than the main memory of a single machine. As a first step, we propose a distributed rule-based CED system called Cloud PARTE, which builds on PARTE [17], a Rete-based CED system. To our knowledge, Cloud PARTE is the first Rete-based CED system that provides the following characteristics:

Distributed, yet unified knowledge base. The Rete nodes are automatically distributed over multiple machines connected in a LAN, yet the semantics of having a single unified Rete network are maintained.

Scalable. Performance degradation from an increasing amount of work can be countered without programming effort, by increasing the number of machines.

Elastic. The system adapts at run time to increases or decreases in the work load. New machines can be added to or removed from a running program and the workers cooperatively redistribute the work.

Dynamically load balanced. The system adapts at run time to change in the distribution of the work load throughout the ruleset. As different subrules become memory- or computationally intensive, the allocation of processing and storage resources is redistributed, across machine boundaries.

2. Context and Requirements

2.1 Traffic Monitoring

One of the potential applications for big data complex event processing is road traffic monitoring. Modern highways are monitored for a wide variety of reasons. Use cases include tracking specific cars for the collection of road tolls, monitoring the overall traffic flow to guide drivers to avoid traffic jams, or the recognition of potentially dangerous situations in order to alert rescue services.

While systems for these applications exist today, they are usually not easily extensible and most of the applications do not provide live feedback. One challenge is to combine the data of various data sources and handle the resulting amounts of data efficiently. Examples for data sources are sensors such as inductive loops,[3] which are permanently embedded into roads to count passing cars, or radar stations used to determine the speed of specific cars. Together with cameras, they can be used to record speed limit violations. Cameras can further be used in combination with on-board units to determine the exact toll for cars as for instance in Germany.[4] Assuming that these data sources can be combined for large and traffic intensive regions or countries in order to facilitate new traffic monitoring applications, the resulting amounts of data outgrows what a single server system can handle.

```
(defrule StationaryCar
  (Car (camera ?c) (plate ?plt) (time ?t1))
  (Car (camera ?c) (plate ?plt) (time ?t2))
  (test (and (> (- ?t2 ?t1) (seconds 30)))
             (< (- ?t2 ?t1) (seconds 60)))
  (Camera (id ?c) (highway ?hw) (position ?p))
=>
  (assert (StationaryCar (camera ?c)
                (plate ?plt) (time ?t1)
                (highway ?hw)  (position ?p))))
```

Listing 1: Example rule to describe a stationary car.

In order to enable a wide range of such applications, rule-based approaches are promising. Forgy [6] proposed the Rete algorithm as an efficient implementation technique. Renaux et al. [17] demonstrated how such an approach could be used for efficient implementation on multicore systems. Imagining an application that uses various input sources to detect traffic jams and accidents, a simple rule can be specified as in the example given in Listing 1. The rule describes how to detect cars that are not moving. The example assumes that the system receives events about cars that include an identifier for the traffic camera recording the car, the license plate information, as well as the time stamp of the event. We assume that a car can be considered as stationary, if it has been recorded at least twice by the same camera in a 30 to 60 seconds timespan. Furthermore, we assume that the system has detailed information about the cameras and their positions on a specific highway. Such information can then be used to trigger a higher-level event `StationaryCar` that can be further processed and used to investigate whether it is an emergency situation or a traffic jam. Unlike existing techniques measuring the flowrate of cars, declarative rules allow reasoning over the causes, e. g., by observing abnormal movement of cars.

Modeling complex traffic monitoring systems based on such an approach requires efficient execution engines. The scenario itself will result in a large amount of rules to de-

[2] *Esper*, EsperTech Event Stream Intelligence, access date: August 15, 2013 `http://esper.codehaus.org/`

[3] *ZELT vehicle monitoring*, Traffic Technology Ltd, access date: August 15, 2013 `http://www.traffictechnology.co.uk/vehicle-monitoring/zelt-loop-detection`

[4] *Toll Collect*, A system for positioning, monitoring, and billing trucks on German motorways, access date: August 15, 2013 `http://www.toll-collect.de/`

scribe the scenarios that are to be detected, and even larger amounts of events coming in from many sensors, at high frequencies. Furthermore, these rules need to be applied to continuous event streams to provide live feedback. Using the Rete algorithm for the rule processing allows for efficient execution, however, it has not widely been used for such large-scale, online scenarios.

2.2 Requirements

In the previous section, we outlined traffic monitoring as an example scenario in more detail, but the domain of complex event detection has a much wider range of applications where live feedback enables individuals and companies to react better and faster to a changing environment. In general, correlating various different data sources provides a much richer context that can be leveraged [1]. These applications have in common is that they follow the *big data* trend, i. e., they are based on the availability of large amounts of data exceeding the capacities of a machine. This *big data* is then mined for correlations and patterns.

For the scenarios we are interested in, data sources produce constant streams of events that need to be processed online to enable live feedback. Rete engines such as Lana–Match[2] enable the use of multiple systems for distributed pattern matching, but they have significant drawbacks such as high latencies, because all matches need to be committed centrally using an transactional concurrency system. Typically, they partition a fact base in order to fit into the memory of a single machine, and then allow parallel execution of the matching process. This approach poses the severe restriction on the supported rules that individual rules may not outgrow the capabilities of a single system, since the granularity of distribution is placed at the rule-level.

The main requirements for continuous complex event processing with big data are as follows. First of all, such a system needs to perform pattern matching **online**, while the events are taking place. Next, it needs to provide support for rules that reason over the whole data set of events, i. e., it needs a **unified knowledge base** that does not make data distribution explicit and does not require any a priory knowledge about how data is stored and processed. Moreover, such a system needs to **scale**: the use of additional machines needs to enable it to compensate for an increase in incoming events. In addition to scalability, it needs to adapt to changing work loads **elastically**: it should be possible to add additional machines to handle increasing work loads at runtime. Finally, it should **dynamically balance load** between machines: the system needs to be able to redistribute data and computation dynamically to avoid overloading machines and to avoid performance bottlenecks. These requirements guide the design of Cloud PARTE, which is discussed in the following section.

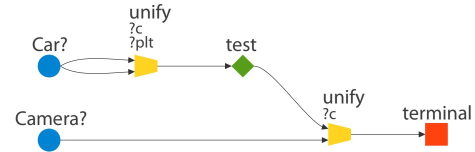


Figure 1: Rete network for the rule defined in Listing 1.

3. Cloud PARTE

Our system is an extension of PARTE, an actor-based, parallel CED system by Renaux et al. [17]. The pattern detection in PARTE builds on the Rete algorithm proposed by Forgy [6]. The Rete algorithm converts a list of rules, such as the one in Listing 1, into a network of nodes, such as shown in Figure 1. The left-hand side of each rule is converted into a set of nodes that perform tests. Incoming events flow through the network as tokens, which are filtered by the nodes through which they pass. Nodes can perform tests on one token, e. g., compare them to a constant, or on two tokens, e. g., test for a unification between two events. In case of a complete match of the left-hand side of a rule, a token will reach the terminal node which represents the right-hand side of the matched rule.

Rete follows a state-saving approach to forward chaining, making it highly efficient for stable fact bases. Since our work is tailored to events, which are stable by design,[5] the Rete algorithm offers a good solution.

Renaux et al. [17] used the strong similarity between a Rete network and a graph of actors communicating via message passing in the design of PARTE by reifying all Rete nodes as actors. In our work, known as Cloud PARTE, we extended the previous system with the means to scale up beyond the bounds of a single machine. Cloud PARTE allows to distribute the detection of complex events over multiple machines, dynamically balancing the load across the available machines. To this effect, the most resource-intensive parts—the actors—are made mobile such that they can be moved around to balance resource usage.

3.1 Architecture

Cloud PARTE shares much of its architecture with its predecessor. Unlike PARTE, which used a custom scheduler and a custom implementation of the actor model to offer soft real-time guarantees in a shared memory context, Cloud PARTE is built on top of the Theron[6] actor library.

Overview From a high level, Cloud PARTE processes events coming from various input sources. A predefined rule set is compiled into a Rete network where each node is represented by an actor to enable parallel execution. These nodes,

[5] Events, after occurring, never change. They have to be stored immutably while useful, and afterwards they expire.

[6] *Theron*, a lightweight C++ concurrency library based on the Actor Model, access date: August 15, 2013 `http://www.theron-library.com`

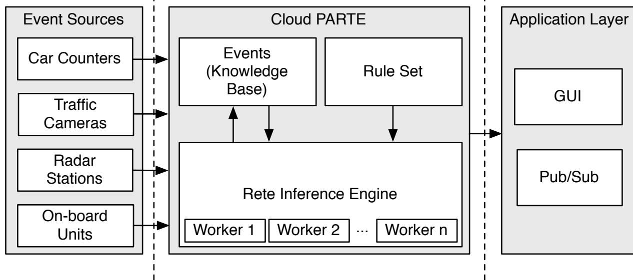


Figure 2: High-level view on Cloud PARTE. Input events from a variety of event sources are processed based on a rule set that is compiled into a Rete network. The nodes of the Rete network are actors which are distributed over a number of worker machines. When a rule matches an event pattern, a higher-level event is generated that can be processed by a user interface or a general publish/subscribe system.

i. e., actors, can be distributed over a number of worker machines. When a rule triggers the match of a certain event pattern, a high-level event is generated, which can then be used by a user interface or a general publish/subscribe system to react appropriately to it. Figure 2 shows a sketch of the overall system.

Distribution The distribution scheme of Cloud PARTE is illustrated in Figure 3. The nodes of the Rete network, represented in the figure as circles, are reified as actors. Multiple actors execute concurrently but maintain a single thread of control internally. The actors may be distributed over multiple machines. Each machine runs a single Cloud PARTE process that can contain many actors, but every actor can only be in one process at any given time. Thus, there is a $1 : 1$ mapping between machines and processes, and a $1 : n$ mapping between processes and actors.

Communication In a directed acyclic graph created by the Rete algorithm, every node has one or more predecessors, and zero, one or multiple successors. In our system, these can be located on the same or on different machines. Actors are identified by unique names, which are used by the Theron framework to route messages to the correct receiver. When the rules are compiled into a Rete network at program startup, actors are given their name and are informed of the names of their predecessors and successors.

Theron uses the Crossroads I/O library[7] to send messages over a network. While this library uses TCP/IP as its communication protocol, guaranteeing delivery of messages, it drops messages when the receivers buffer is full. Unlike in some Belief-Desire-Intent multiagent systems such as Jason [5], Cloud PARTE's semantics do not allow actors to fail or refuse to react to messages. Our system hence employs a best-effort based approach to overcome this issue, by acknowledging the receipt of a message with a confirmation message, similar to TCP. For this purpose, every Cloud PARTE process contains one "RemoteSender" actor, which is responsible for the confirmation protocol, and propagates deserialized messages to the correct destination by local[8] message-send. Despite being a main implementation issue, we do not consider this a scientific challenge, since it could be overcome using existing techniques.

[7] *Crossroads I/O*, A socket library providing platform-agnostic asynchronous message-sending across different network protocols, access date: August 15, 2013 `http://www.crossroads.io`

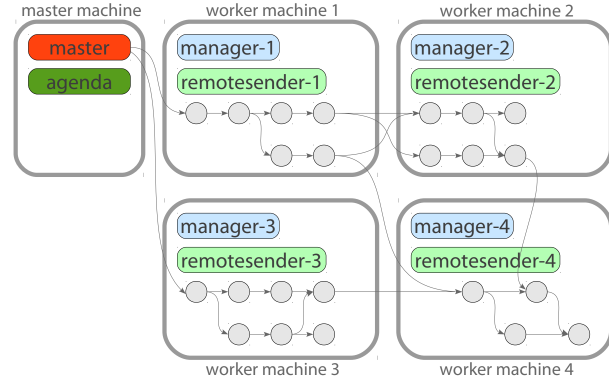


Figure 3: The distribution scheme employed by the system. The boxes represent machines: the single master machine contains the master actor and the agenda actor, and every worker machine contains a part of the Rete network and a manager actor.

Coordination While Cloud PARTE is distributed, it is not decentralized. A *master* actor oversees the entire system, and knows where each node is at every moment (assuming no failure happened, failure resilience is considered future work). During initialization, the master actor parses the user-defined rules, creates a Rete network out of them, decides which machines the nodes should move to, and informs those machines' *manager* actors of the nodes they need to spawn. A manager actor exists per process. They each oversee the part of the Rete network located on the machine they are running in, and can spawn – and kill – Rete nodes and the RemoteSender within their process. They keep track of the nodes living in their process, and can for instance determine whether the destination of a message-send is local.

The master actor's process further contains the agenda actor. The agenda sequentializes side-effects performed from within the Rete network. More specifically, whenever a complete match for a rule is found, any callback function specified as the consequence of the rule is to be called through the foreign function interface. This execution should not be performed locally in the Rete node that detected the match. Instead, a message requesting the execution is sent to the agenda actor, which executes it on the master machine.

[8] We refer to actors in the same process as *local actors*, and actors in other processes as *remote actors*.

3.2 Distributed Execution Model

The Theron library implements the actor model by providing an `Actor` class that can be extended to create new types of actors. Within a process, a Theron instance contains many actors that run concurrently on a thread pool. On each machine in a LAN, one process is started, and their Theron instances are connected. Inside an actor no parallelism is used.

A frequent issue in such concurrent systems is contention for shared resources. Even with the actor model taking care of low-level data races, access to actors' inboxes remains a point of contention. Where PARTE offered lock-free inboxes, Theron's need to potentially serialize messages across a LAN require it to forgo this optimization. Within the Rete graph, though, every node has only one or two predecessors. Only the master actor, the agenda and the RemoteSender are heavily contended. Though these do pose as potential bottlenecks, our experiments showed they are able to keep up with the rest of the system. Furthermore, the RemoteSender is only present to overcome a technical issue of the Theron library. The only real, inherent bottlenecks lies in the communication over the LAN. Reducing this network traffic is part of our future work. One approach we envision is to distribute the Rete graph in such a way that strongly connected subgraphs in the Rete network tend to be physically close. This would reduce the need for inter-process communication, and with that the contention for the LAN.

A second issue systems like Cloud PARTE face is efficient serialization. When a message is sent to a local actor, Cloud PARTE makes use of the shared memory space to skip serialization. When the receiving actor is remote, however, pointer-indirections would be invalid, so the message is serialized to a flattened, uniform representation. When choosing the serialization format, two considerations were made: the compactness of the serialization, and the speed at which the (de-)serialization can take place. These have an impact on respectively the required bandwidth and the incurred latency of message sends. In our experiments, JSON[9] revealed sufficiently small and fast, though we do consider improving serialization future work.

Despite its support for distribution, Theron does not offer support for code mobility, i. e., the ability to dynamically change the bindings between code fragments and the location where they are executed [7]. Since we require actors to migrate between machines, we built mobility on top of the existing library. Since an actor maintains no runtime stack across turn borders, it can be migrated by sending its inbox and its internal state.

We devised the schema depicted in Figure 4 to deal with messages sent to an actor while it is moving. In a first phase, the actor to move informs its predecessors, which from that moment on start buffering their messages for the migrating actor. Once all predecessors are buffering, the actor is serialized and destroyed, and the serialized representation is sent to the destination machine's manager. There, it is deserialized and reconstructed, and the predecessors are notified so that they can send their buffered messages.

[9] JSON (JavaScript Object Notation) is a text-based standardized language used for data interchange. It is human-readable, which eases debugging but does not lead to the most compact representation possible.

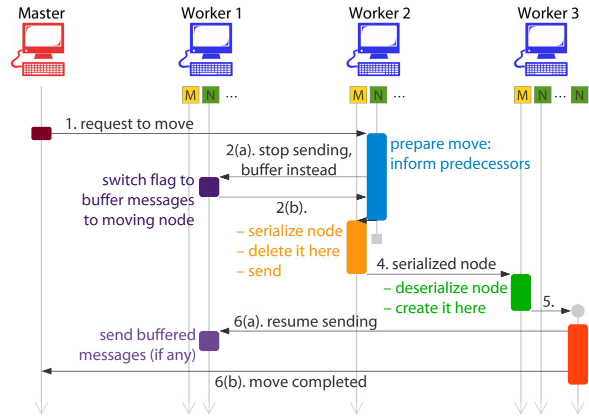


Figure 4: The procedure to move a node to a different machine. Every worker machine contains one manager actor (M) and several Rete node actors (N). In this figure, a node on worker 2 moves to worker 3. The node on worker 1 is a predecessor of the moving node: when it receives message 2(a) it starts buffering its messages destined for the moving actor, when it receives message 6(a) it sends the buffered messages to the now moved node.

3.3 Load Detection and Balancing

Cloud PARTE uses a global, centralized load balancing strategy [19]. The manager of each process measures the load of its local actors on a regular interval, and relays that information to the master actor. The master then makes load balancing decisions based on this global information.

Non-trivial heuristics to base the load balancing decisions on are outside of the scope of this paper, and considered future work. Our experiments described later in this paper were conducted with a simple heuristic that defines load as a threshold on the number of unprocessed messages in an actor's inbox. Machines are in turn labeled as overloaded or underloaded based on the amount of overloaded actors they contain. Whenever at least one overloaded and one underloaded machine are found, the master actor selects an overloaded actor on the most overloaded machine and migrates it to a randomly selected underloaded machine.

The movement of actors is fully transparent: the user does not need to worry about the location of actors, or when and where to move them. The load balancing algorithm determines this automatically. The predecessors of the moved actor need not do any lookup to 'find' the actor at its new location: the unique name of the actor remains valid.

4. Evaluation

4.1 Methodology and Setup

For the evaluation of Cloud PARTE's performance, we conducted a series of microbenchmarks. They ran on a network of commodity computers, each containing a quad-core processor[10] and 8 GB of RAM memory and running the Ubuntu 12.04.2 operating system with a Linux 3.2.0-43-generic kernel. All code was compiled using gcc 4.6.3 with optimization flag -O3. The network is rated at 1000 Mbit/s. Based on the methodology proposed by Georges et al. [8], every configuration is executed at least 30 times, and is executed additionally until either a confidence level of 95% is achieved, or the configuration has been executed 75 times.

The experiments are parametrized by the following three dimensions:

- The number of machines used. To measure the scalability of the system, we gradually increased the number of machines and measured the effect on the throughput. In an ideal case, doubling the amount of machines doubles the throughput.
- With/without confirmation messages. Enabling confirmation messages is expected to cause a decrease in performance, because *1*) for every received message a confirmation is sent and received, *2*) for every received message a check for duplication is performed, and *3*) for every sent message it is regularly checked whether a confirmation has already been received or whether a resend is warranted. To demonstrate that we incur a performance-hit from the inherently hard problem of verified message-delivery in a distributed system, we compare the version of Cloud PARTE utilizing the RemoteSender with a version assuming failure-free communication channels.
- With/without dynamic load balancing. The load balancing scheme introduced in subsection 3.3 is expected to increase efficiency. We measured this by comparing the system with dynamic load balancing with a version of the system where dynamic load balancing was disabled.

The benchmarks we performed to evaluate Cloud PARTE are a combination of benchmarks defined earlier by Renaux et al. [17] for the evaluation of PARTE, and a new benchmark measuring the newly added aspects, namely distribution and dynamic load balancing. The existing benchmarks can be categorized as *1*) a number of microbenchmarks employing only pipeline parallelism while conducting *simple*, *complex*, or *heavyweight* tests; *2*) a number of microbenchmarks executing multiple of those pipelines in parallel; *3*) a microbenchmark executing a *tree*-shaped Rete-network that is mostly unification- and communication-heavy; and *4*) a microbenchmark where multiple rules require a lot of processing, but only one rule triggers the benchmark's end, such

[10] Specifically, an Intel Core i5-3570, a processor with a 64-bit architecture running at a clock frequency of 3.40 GHz, with 6 MB of on-chip cache.

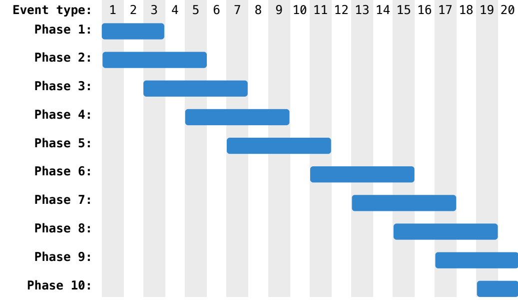


Figure 5: Partitioning of benchmark into phases: 100,000 events of 20 types are asserted in 10 phases, gradually progressing through these types. Each phase asserts events from three to five different but consecutive event types.

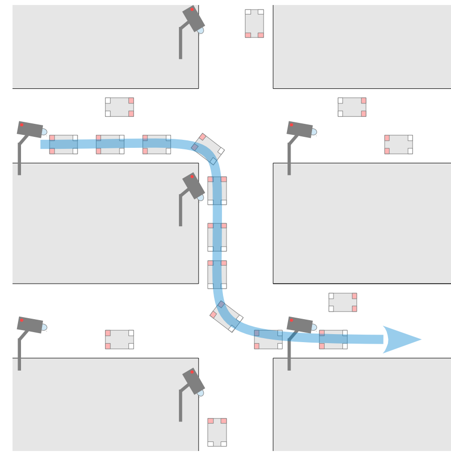


Figure 6: A scenario that could lead to a situation such as depicted in Figure 5: a number of cars move through an area, within view of multiple cameras. As they move, they go out of view of one camera and move into view of another.

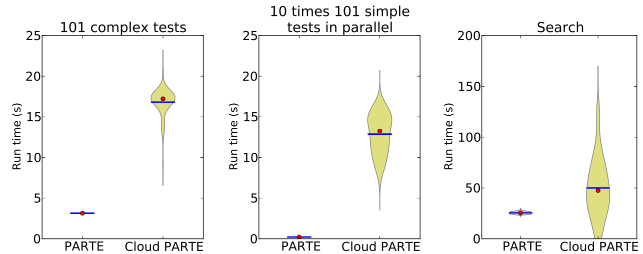
that non-FIFO scheduling may cause a superlinear improvement in detection latency. The new benchmark activates the rules in *phases*, as demonstrated in Figure 5, simulating a scenario such as the one depicted in Figure 6 where a group of tracked cars moves though the view of multiple sensors. In general, a scenario where different parts of the Rete network become computation and/or memory-intensive during the run time of the CED system is what caused the need for load balancing in Cloud PARTE. Hence, this benchmark validates the ability to adapt to changing workloads.

For every configuration, i. e., for every run of the benchmark with a certain set of parameters chosen, we measured one of two variables:

Run time The time between inserting the first event of the benchmark into the system, and successfully having processed all of them. In other words, the experiments measure the time it takes for all of the events to percolate through the Rete network.

Benchmark	Run time (ms) PARTE	Run time (ms) Cloud PARTE	Slow-down
101 simple tests	21.7± 1.4	391.4± 10.0	18.04
501 simple tests	70.6± 1.1	2,053.6± 53.9	29.10
101 complex tests	3,145.0± 4.3	16,810.4± 375.0	5.35
. . . with variables	7,302.1± 3.0	38,111.1±1,117.2	5.22
10×101 simple tests	212.8± 0.6	12,875.7± 550.8	60.49
10×8 heavy tests	4,907.8± 2.8	11,497.0± 580.9	2.34
16 heavy tests	104.4± 1.0	379.9± 8.1	3.64
32 heavy tests	199.4± 1.0	750.6± 13.5	3.76
64 heavy tests	389.2± 1.0	1,432.3± 31.5	3.68
128 heavy tests	772.7± 1.8	2,631.9± 81.2	3.41
Joining tree	29.0± 0.4	173.9± 2.5	5.99
Search	25,526.9±264.3	49,979.1±6,497.7	1.96

Table 1: Comparison of Cloud PARTE's and PARTE's run times, and the 95% confidence interval of the results.



(a) Runtime of the "101 complex tests" benchmark (b) Runtime of the "10 × 101 simple tests" benchmark (c) Runtime of the "search" benchmark

Figure 7: A comparison of the run time of PARTE and Cloud PARTE when executing a subset of the benchmarks on a single machine.

Throughput The number of events that can be processed by the system per time unit. The throughput is measured indirectly, by dividing the number of events generated during the benchmark by the total run time.

Results are visualized in beanplots [14], a variation on boxplots. The width of a 'bean' at a given height indicates the density of measured results with that run time or throughput. The horizontal lines indicate the arithmetic mean of the results, and the dot marks the median.

4.2 Efficiency

The first experiment we conducted consists of a comparison of Cloud PARTE to PARTE, using the benchmarks created for PARTE, run on one machine. Cloud PARTE was set up with one worker process running on the same machine as the master process. As expected, Cloud PARTE did considerably worse, requiring inter-process communication for I/O, and constantly depending on locking access to inboxes, whereas PARTE uses nonblocking data structures, and heavily exploits the shared memory architecture. Furthermore, in PARTE benchmarks end when a match is found, whereas in Cloud PARTE a trip to the master process is required first. Still, these benchmarks serve as a useful baseline, demonstrating Cloud PARTE's overhead.

Figure 7a shows the situation for the representative case of the "101 complex tests" benchmark. Cloud PARTE is on average 5.35 times slower than PARTE. The worst situation arose with the "10 times 101 simple tests in parallel" benchmark, an exceptionally computation-light and hence scheduling- and communication-heavy benchmark. As Figure 7b shows, an average slowdown of 60.49 was measured. The best-case, where Cloud PARTE is only 1.96 times slower, can be found in the "search" benchmark, whose results are plotted in Figure 7c. For completeness, Table 1 shows the run times of all benchmarks on both systems. Note how PARTE, being tailored towards soft real-time behavior, has much less variation in its run times – even when normalized to absolute run time.

4.3 Static Scalability

The second experiment assesses the scalability of Cloud PARTE by measuring the throughput as the amount of machines is increased. Two variants of Cloud PARTE are compared, one using confirmation messages and one without.

In this benchmark, a rule duplicates every incoming event to fifteen pipelines conducting sixteen heavy tests each on the event, and each generating a complex event on completion, signaling a seventeenth and final rule, which unifies all complex events and ends the benchmark when all events were processed. As an initial setup, this Rete network is deployed over eight machines and run. Subsequently, the number of machines is halved (to four), by merging two machines' processes into one; this is repeated two more times (for set-ups with two and one machine). For these tests, dynamic load balancing was disabled.

Table 2 shows the results. The version with confirmation messages performs, as can be expected, less well than the one without them: on eight machines the extra communication overhead of the confirmation messages nearly halves the speed of the system. The table shows that moving from a set-up with one machine to one with two machines does not provide much benefit: this is because in the case of one machine all communication is local while when using two machines (de)serialization and network communication is necessary. However, the throughput nearly doubles when increasing the amount of machines from two to four, and from four to eight. While ideal speedup is not achieved, from four machines on, Cloud PARTE achieves a higher throughput than PARTE, demonstrating that an automatically load balanced distribution based on mobile actors enables scaling of inference engines beyond the capabilities of a single machine.

4.4 Dynamic Scalability

In the third experiment, the benefits of dynamic load balancing and elasticity are demonstrated. To this end, three scenarios are compared: one in which dynamic load balancing is disabled, secondly a scenario that allows dynamic load

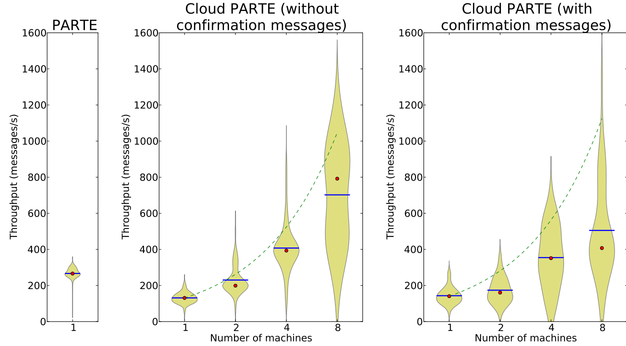


Figure 8: Beanplots indicating the distribution of measurements of the throughput (in messages per second) of PARTE, Cloud PARTE without confirmation messages, and Cloud PARTE with confirmation messages, as the amount of machines increases. The green dashed lines indicate ideal speedup.

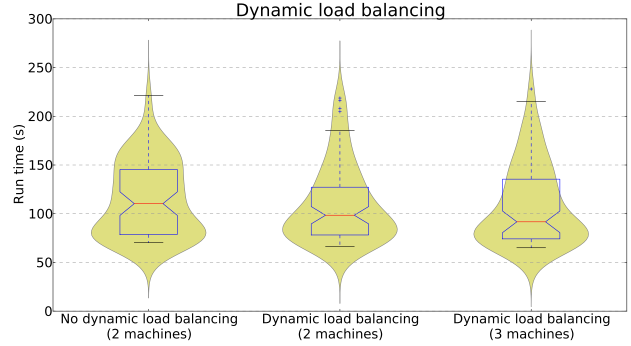


Figure 9: Comparison of run time with manual distribution of the Rete network over 2 machines and (a) no load balancing, (b) load balancing, (c) load balancing and an extra third machine. The blue +s indicate outliers.

Number of processes	PARTE	Cloud PARTE	
		no conf.	confirmation
1	266 (1.00x)	131 (0.49x)	141 (0.53x)
2	—	199 (0.75x)	161 (0.60x)
4	—	393 (1.48x)	351 (1.32x)
8	—	792 (2.98x)	407 (1.53x)

Table 2: Median throughput and ratios of the benchmark of Figure 8.

balancing, and lastly a configuration in which an additional empty machine is present. In all scenarios the Rete network is initially distributed over two machines at startup by a hard-coded distribution scheme. The second scenario shows the benefit of dynamic load balancing, whereas the third shows the elasticity provided by the system.

The most relevant benchmark is hence the new 'phased' benchmark introduced in subsection 4.1 and depicted in Figure 5, which consists of twenty rules – one for each event type. The initial distribution of all twenty rules on the first machine corresponds to a reasonable real-life situation, since one may not know in advance how the entities whose events are being pattern matched will behave.

The results plotted in Figure 9 show that both the dynamic load balancing and the elasticity of Cloud PARTE can offer measurable improvements. The median run time for the base case where load balancing is disabled is 110.36 seconds, while the median time for the load balanced version is 98.45 seconds, i. e., 10.8% faster. The addition of a third machine decreases the run time even further to 91.69 seconds, a 16.9% decrease compared to the original.

By logging the migrations triggered by the load balancer, we confirmed that the performance increase indeed correlates to the correct migration of overloaded nodes. The heuristics used by the load balancing algorithm described in subsection 3.3 are little more than a proof-of-concept, and improving these is considered future work. Still, it tends to move nodes around such that the active nodes of the current phase are balanced over the machines. Occasionally, it moves too few or too many nodes, since the number of nodes which have moved are simply not part of the equation the load balancer uses. In those cases, the run time peaks.

4.5 Discussion

We conclude from our performance evaluation that mobile actors, as employed in Cloud PARTE, succeed in offering an improvement over the state of the art in complex event detection. While an overhead definitely but obviously exists in the distribution, this overhead can be overcome by increasing the number of machines. An additional benefit is gained from load balancing, even with the simple heuristics and a straightforward migration system which only allows one node to be on the move at the same time, and uses the suboptimal JSON format for serialization.

The elasticity of Cloud PARTE effectively manages to provide 'free' additional speedup by simply plugging in another machine. Currently, all machines have to be present at startup time of Cloud PARTE, though this is but a technical issue: there is no conceptual limitation prohibiting machines to arrive when the system is already running. Even support for unloading machines and subsequently retiring them from the system is just some engineering-effort away.

A number of open problems remain, which require more research to be resolved. Those are listed in section 6.

5. Related Work

5.1 Parallel and Distributed Rete

The Rete algorithm is widely used, and several parallel and distributed variants exist already.

The parallelization approach taken by Gupta et al. [10] is similar to ours: every node of the Rete network is considered a 'task', and is given an internal thread of control. Based on

the data-flow like organization of the Rete network, multiple tokens will be flowing through the Rete network simultaneously, and as a result multiple nodes can be active at the same time. Their approach is more general, as it's not limited to event processing, but they therefore have to forgo conceptual optimizations that rely on temporal reasoning, e. g., in Cloud PARTE events expire automatically, whereas in their approach facts need to be explicitly retracted.

The approach of Kelly and Seviora [15] increases the granularity of the parallelization: instead of considering individual Rete nodes as tasks, single token–token comparisons are parallelized. Thus, nodes of the Rete network are split into several copies, one copy for every token in their associated memory node. Each of these copies is a separate task. The level of parallelism increases, but the amount of communication increases as well; as such this approach is suitable for shared memory systems but not for distributed systems.

A different approach to distributing the Rete algorithm is taken by Aref and Tayyib [2]: in their Lana–Match algorithm the Rete network is duplicated among a number of worker processors. Changes to the working memory made by the workers are synchronized by a single, centralized master, using an optimistic algorithm which backtracks in case of conflicting updates. The optimistic algorithm assumes each rule will only change a very small part of the fact base and that conflicts are rare. Furthermore, in Lana–Match the centralized master processor will be processing all changes to the working memory, and can therefore become a bottleneck, while in Cloud PARTE the master only handles incoming data and the working memory is spread over the workers. In Cloud PARTE, the event base is spread over the worker machines, but appears as a single unit.

Lastly, yet other systems use a hierarchical blackboard, for example the Parallel Real-time Artificial Intelligence System (PRAIS) by Goldstein [9] and the Hierarchically Organized Parallel Expert System (HOPES) by Dai et al. [3]. In these systems, each knowledge source (KS) contains a rule, and multiple KSs can simultaneously process information. This information is shared through a blackboard, a global database that contains all information about the problem currently in the system. KSs incrementally update the blackboard leading to a solution to the problem. This approach is more coarse grained than Cloud PARTE, leading to a lower level of parallelism. These systems also do not focus on scalability and elasticity.

5.2 Mobile Actors and Dynamic Load Balancing

Using mobile actors as a mechanism for load balancing exists in other systems. For instance, Lange et al. [16] introduce "Aglets", a Java implementation of mobile actors (also called mobile agents). Fuggetta et al. [7] describe a distributed information system, in which information is dispersed throughout a network, and code moves to be 'closer' to the data it uses. In these systems, the actors are proactive: they decide when to move and to where, for instance when it is impossible or unfeasible to move the information (e. g., because of its huge size). In our system movements are reactive: the load balancer decides when an actor should move, to provide each actor with the computational resources it needs.

SALSA [18] is an actor-based language in which actors are distributed over multiple machines. Actors are serialized for migration, as in our system, and references between actors remain valid through the use of a universal naming scheme, this too is similar in our system. In SALSA, the code of a universal actor needs to be present on every machine, in our system this is partly the case: the code to create any of the five primitive node types of the Rete algorithm needs to be present on each machine, but the code supplied by the user as part of a rule will be migrated along with the actor that contains it (and it is reparsed when the actor is resumed). Desell et al. [4] describe how load balancing is added to SALSA. A major difference between Cloud PARTE and SALSA is that, in Cloud PARTE, the load balancing is centralized (a master machine gathers load information and makes load balancing decisions), while SALSA supports several decentralized load balancing techniques. In these techniques, lightly loaded machines will attempt to steal work from other machines. These ideas are applicable to Cloud PARTE and will be investigated in the future.

Lastly, in the blackboard systems discussed in the previous subsection (PRAIS [9] and HOPES [3]), each actor contains one rule, while in Cloud PARTE each actor contains only a part of the rule (one node of the Rete network). As a result, load balancing in these blackboard systems would consist of moving a complete rule, while in Cloud PARTE parts of rules can be executed on different machines and move independently. Because of this finer granularity, Cloud PARTE will allow a higher amount of parallelism and greater load balancing flexibility.

6. Conclusions and Future Work

Cloud PARTE is a complex event detection system that distributes the Rete algorithm over multiple machines, using mobile actors in order to respond to changing work loads. It is designed for applications such as traffic monitoring or crowd management, where large amounts of data need to be processed and it is therefore necessary to use multiple machines. Cloud PARTE does not require programmer effort in the distribution of rules, but despite the transparency of the load balancing and elasticity, it retains the semantics of a single unified Rete network.

Variations in the work load make dynamic load balancing necessary. Cloud PARTE uses mobile actors, i. e. actors that can migrate between machines at run time, to provide load balancing. This also enables elasticity: a machine can be added at run time and the load will be redistributed.

Based on a set of micro- and synthetic benchmarks, we show that Cloud PARTE is a scalable and elastic system that

can handle increasing amount of work, and that dynamic load balancing provides an additional benefit in distributing the work evenly over the available machines.

Future work will address the large communication overheads, and improve the load balancing algorithm to take into account more factors, such as putting closely connected Rete nodes on the same machine, or estimating the cost and potential gain of the migration. Furthermore, migration currently is a cooperative effort, requiring both the migrating actor and its predecessors to work in close collaboration. Future research should focus on decoupling the actors, up to the point of adding fault tolerance to the system, such that failure of individual machines can be overcome without undermining the entire system. Finally, we envision extensions to the expressiveness of the querying system in the form of dynamic queries of the knowledge base, and support for negation and existential quantification in the rule language.

References

[1] S. S. Adams, S. Bhattacharya, B. Friedlander, J. Gerken, D. Kimelman, J. Kraemer, H. Ossher, J. Richards, D. Ungar, and M. Wegman. Enterprise context: A rich source of requirements for context-oriented programming. In *Proceedings of the 5th International Workshop on Context-Oriented Programming*, COP'13, pages 3:1–3:7, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2040-5. .

[2] M. M. Aref and M. A. Tayyib. Lanamatch algorithm: a parallel version of the retematch algorithm. *Parallel Computing*, 24(5-6):763–775, June 1998. ISSN 0167-8191. .

[3] H. Dai, T. Anderson, and F. Monds. On the implementation issues of a parallel expert system. *Information and Software Technology*, 34(11):739–755, November 1992. ISSN 09505849. .

[4] T. Desell, K. E. Maghraoui, and C. Varela. Load Balancing of Autonomous Actors over Dynamic Networks. In *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, number 9 in HICSS'04, pages 1–10, 2004. ISBN 0769520561.

[5] A. Fernández Díaz, C. Benac Earle, and L.-A. Fredlund. Adding distribution and fault tolerance to jason. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, AGERE! '12, pages 95–106, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1630-9. .

[6] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1): 17–37, 1982. ISSN 0004-3702. .

[7] A. Fuggetta, G. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5): 342–361, May 1998. ISSN 00985589. .

[8] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76. ACM, 2007. ISBN 978-1-59593-786-5. .

[9] D. Goldstein. Extensions to the Parallel Real-Time Artificial Intelligence System(PRAIS) for fault-tolerant heterogeneous cycle-stealing reasoning. In *Proceedings of the 2nd Annual CLIPS ('C' Language Integrated Production System) Conference, NASA. Johnson Space Center*, pages 287–293, Houston, September 1991.

[10] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel algorithms and architectures for rule-based systems. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 28–37. IEEE Computer Society Press, 1986. ISBN 0-8186-0719-X. .

[11] L. Hoste, B. Dumas, and B. Signer. Mudra: A Unified Multimodal Interaction Framework. In *Proceedings of ICMI 2011, 13th International Conference on Multimodal Interaction*, Alicante, Spain, Nov. 2011.

[12] L. Hoste, B. De Rooms, and B. Signer. Declarative Gesture Spotting Using Inferred and Refined Control Points. In *Proceedings of ICPRAM 2013, 2nd International Conference on Pattern Recognition Applications and Methods*, Barcelona, Spain, February 2013.

[13] M. W. Kadous. Learning Comprehensible Descriptions of Multivariate Time Series. In *Proceedings of ICML 1999*, Bled, Slovenia, June 1999.

[14] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, Oct. 2008. ISSN 1548-7660.

[15] M. A. Kelly and R. E. Seviora. A multiprocessor architecture for production system matching. In *Proceedings of the National Conference on Artificial Intelligence*, pages 36–41, 1987.

[16] D. Lange, M. Oshima, G. Karjoth, and K. Kosaka. Aglets: Programming mobile agents in Java. In *Proceedings of the International Conference on Worldwide Computing and Its Applications*, pages 253–266, 1997. .

[17] T. Renaux, L. Hoste, S. Marr, and W. De Meuter. Parallel gesture recognition with soft real-time guarantees. In *Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions*, SPLASH '12 Workshops, pages 35–46, New York, NY, USA, October 2012. ACM. ISBN 978-1-4503-1630-9. .

[18] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20, Dec. 2001. ISSN 03621340. .

[19] M. J. Zaki, W. Li, and S. Parthasarathy. Customized Dynamic Load Balancing for a Network of Workstations. Technical report, The University of Rochester, Rochester, New York, USA, 1995.