Vrije Universiteit Brussel

Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Large-scale pattern recognition

Dynamic load balancing using code mobility

Dissertation submitted in partial fulfillment of the requirements for the degree of
Master in Engineering: Computer Science

Janwillem Swalens

Promotor: Prof. Dr. Wolfgang De Meuter
Advisors: Thierry Renaux
Lode Hoste
Dr. Stefan Marr

June 2013

Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Departement Computerwetenschappen
Software Languages Lab

Grootschalige patroonherkenning

Dynamische belastingsverdeling door middel van mobiele code

Proefschrift ingediend met het oog op het behalen van de titel
Master in de Ingenieurswetenschappen: Computerwetenschappen

Janwillem Swalens

Promotor:	Prof. Dr. Wolfgang De Meuter
Begeleiders:	Thierry Renaux
	Lode Hoste
	Dr. Stefan Marr

Juni 2013

Abstract

Applications such as traffic monitoring, crowd management or gesture recognition systems intend to recognize patterns in streams of data captured by their sensors, and react to them accordingly. *Large-scale pattern recognition* is concerned with the recognition of complex patterns in these data-intensive streams. Programming these systems is hard, which is why a rule-based approach is preferable: a declarative language allows the programmer to define rules that describe patterns, and how to react to them.

As both the sensor hardware advances, e.g. more cameras are used or they support higher resolutions, and more complex patterns need to be recognized, the amount of data and the amount of rules and their complexity will increase. This poses a problem: how do we scale our system to handle this increasing complexity? Existing solutions for rule-based systems are limited to a single machine or require manual programming to scale.

The solution proposed in this thesis is to distribute the work over multiple machines. Based on PARTE (Parallel Actor-based ReTe Engine), which parallelizes the Rete algorithm using an actor-based approach, a system is created in which these actors are distributed over the available machines: DPARTE (Distributed PARTE).

The system uses dynamic load balancing to redistribute the work while the program is running. This is done by introducing code mobility into the system: the actors become "mobile actors", they will move at run time to distribute the work load evenly.

Based on an experimental evaluation, we substantiate our claims that distributing the work over multiple machines leads to a scalable and elastic system that can handle increasing amounts of work, and that dynamic load balancing provides an additional benefit in distributing the work evenly over the available machines.

To our knowledge, DPARTE is therefore the first Rete-based rule engine with a unified fact base that supports automatic distribution and dynamic load balancing.

Samenvatting

Toepassingen zoals systemen voor verkeersmonitoring, crowd control of gebarenherkenning, herkennen patronen in gegevensstromen die worden geregistreerd door hun sensors, en reageren hierop. *Grootschalige patroonherkenning* is de herkenning van complexe patronen in deze grote gegevensstromen. Het programmeren van deze systemen is moeilijk, en daarom wordt een aanpak gekozen gebaseerd op regels: een declaratieve taal laat de programmeur toe regels te definiëren die patronen beschrijven, en hoe erop te reageren.

Naarmate zowel de sensorhardware verbetert, bvb. doordat meer camera's worden gebruikt of deze hogere resoluties ondersteunen, en meer complexe patronen moeten herkend worden, zal de hoeveelheid data en het aantal regels en hun complexiteit stijgen. Dit stelt een probleem: hoe schalen we een systeem zodat het met deze stijgende complexiteit om kan gaan? Bestaande oplossingen beperken zich tot één enkele machine of vragen handmatig werk om te kunnen schalen.

Deze thesis stelt voor om het werk te verdelen over meerdere machines. Gebaseerd op PARTE (Parallel Actor-based ReTe Engine), dat het Rete-algoritme parallelliseert door middel van actors, wordt een systeem ontworpen waarin deze actors verdeeld worden over de beschikbare machines: DPARTE (Distributed PARTE).

Het systeem maakt gebruik van dynamische load balancing om het werk te herverdelen tijdens de uitvoering van het programma. Dit wordt gerealiseerd door "mobiele code" toe te voegen aan het systeem: de actors worden mobiel, wat wil zeggen dat ze kunnen verplaatst worden tijdens de uitvoering van het programma.

Gebaseerd op een experimentele evaluatie staven we de beweringen dat het verdelen van werk over meerdere machines leidt tot een schaalbaar en elastisch systeem, en dat dynamische load balancing een voordeel biedt door ervoor te zorgen dat het werk gelijk verdeeld is over de machines.

Voor zover wij weten is DPARTE het eerste regelgebaseerde systeem gebruik makend van Rete met een verenigde feitendatabank dat automatische distributie en dynamische load balancing ondersteunt.

Acknowledgements

This thesis would not have been possible without the many people that helped me along the way.

First of all, I would like to thank my supervisors: Thierry Renaux, Lode Hoste, and Stefan Marr. They sacrificed many hours guiding me, discussing problems I encountered, providing invaluable comments and insights, and proofreading this document. Each time, the weekly meeting provided me with new ideas and plans. Without them, this thesis would not exist today.

I would also like to thank Prof. Dr. Wolfgang De Meuter for promoting this thesis, as well as for giving me the opportunity to create it at the Software Languages Lab.

The complete team of the Software Languages Lab deserves my thanks for their comments and feedback, for instance during the intermediate presentations. They gave me the education that brought me here.

Finally, I would like to thank my family, for providing me a nice and welcoming home, and for their continuous support and concern. I am sure they will try their best to read and understand this thesis. Bedankt mama, papa, en Maud.

Janwillem Swalens

Contents

List of figures

Chapter 1
Introduction

Applications such as traffic monitoring, crowd management or gesture recognition systems rely on pattern recognition to process large streams of data and extract meaningful events from them. As on the one hand sensor hardware advances and on the other hand applications want to recognize more complex patterns, the amount of processing power and memory required by pattern recognition system increases. To scale these systems, we propose to use multiple machines and distribute the work over them. Furthermore, to take into account variations in the type of work during the execution of the program, dynamic load balancing is used to redistribute the work at run time.

Context

Several major developments have been made in the last few years in the area of human–computer interaction. Powerful hardware – such as the Microsoft Kinect 3D camera, the Nintendo Wii with a motion detecting controller, smartphones and tablets with multi-touch displays, and a multitude of other sensors – has become a commodity, easily available for the average customer.

As a result, an increasing number of programs using this hardware have been created. Programmers invent new ways in which this hardware can be applied, e.g. they make user interfaces more intuitive (Google Earth using multi-touch gestures), they add intelligence to their programs (Google Now uses a smartphone's sensors such as its GPS to display relevant information) or they design creative new games (dancing games powered by the Kinect or bowling using the Nintendo Wii).

Under the hood, these programs use *pattern recognition* to recognize patterns in the data collected by sensors. Pattern recognition focuses on "how machines can observe the environment, learn to distinguish patterns of interest from their background, and make sound and reasonable decisions about the categories of the patterns" [Jain and Duin, 2000]. The aim of pattern recognition is to identify or *classify*

objects based on their characteristic *features* [Theodoridis and Koutroumbas, 2009]. A *pattern* can be seen as anything that characterizes or represents objects.

For example, consider a multi-touch interface such as the screen on a smartphone or tablet. The user can touch the screen, and the hardware will provide us with the (x, y) coordinates of the touched points on the screen. Some patterns such a system might want to recognize in the data are pinch, stretch or rotate gestures (Figure 1.1).

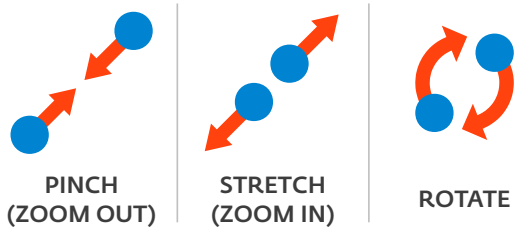


Figure 1.1: Pinch, stretch, and rotate gestures, used on multi-touch devices to manipulate images.

Related to this is the field of *complex event detection* (CED). Complex event detection processes or analyzes streams of data (such as the images captured by a Kinect) and tries to detect events in them (such as a waving user). The application can then respond to these events in an appropriate way.

Take for instance a dance game using the Microsoft Kinect, where the player has to imitate a dance shown on the screen. An overview of the architecture of such a system is shown in Figure 1.2: a set of Kinect cameras capture streams of images in 3D, and the software aims to recognize events – such as the Y, M, C and A movements in the popular song imitated by the player. The 3D images captured by the hardware are processed into skeletons of 3D points, and a rule engine detects events in this data. One such event might be "hands up", the application can then react to these events by rewarding the player with points.

Similarly, on a large festival there might be a number of cameras spread over the festival site (Figure 1.3). These generate streams of data: images of different areas on the site. Software using complex event detection can be used to detect events such as a large crowd moving from one area to another, or a dangerous situation such as a panic outbreak or a fire. The software can then notify a control center which can take the necessary actions.

Common to the use case scenarios studied in this thesis is that the streams captured by the hardware are *data-intensive*, i.e. they contain a lot of data and require a lot of processing. *Large-scale pattern recognition* is concerned with the recognition of patterns on these data-intensive streams.

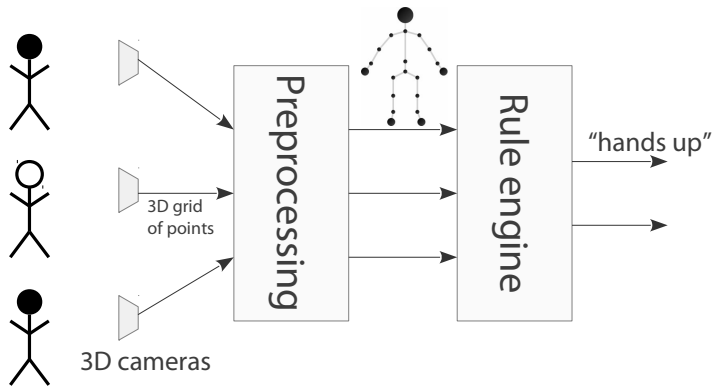


Figure 1.2: Overview of a CED system that recognizes gestures captured by 3D cameras.

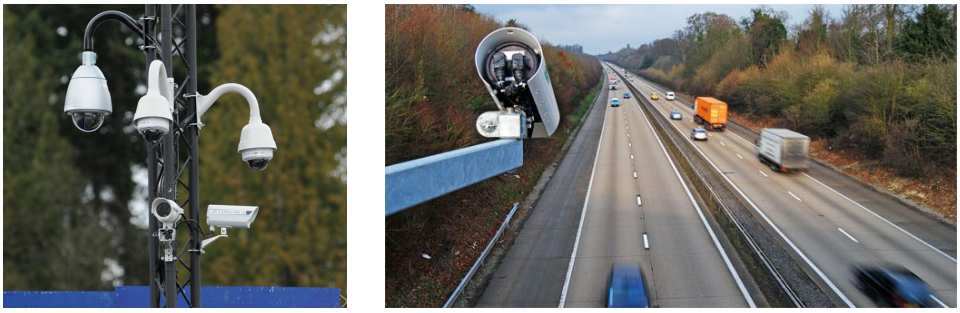


Figure 1.3: Surveillance cameras at an event such as a festival, and in traffic.

Large-scale pattern recognition poses some unique problems, in particular related to how the system can cope with these large amounts of data. More processing power and more memory will be required for these systems, which poses problems distinct from those encountered in regular pattern recognition.

Problem statement and approach

In complex event detection streams of data are analyzed to detect events. As hardware continues to develop (e.g. an increase in resolution or a higher sampling rate), it generates more data. At the same time, applications want to detect more and increasingly complex patterns in the data. In other words, the complexity of the problem increases greatly.

To handle the increasing amount of data a pattern recognition system has to process, and the increasing complexity of patterns it should recognize, it will need more pro-

cessing power and memory. Previous work argues that current systems approach and exceed the limits of processing power and memory available on a single machine [Renaux, 2012].

This dissertation proposes to distribute the workload over multiple machines. A system based on the Rete algorithm [Forgy, 1982] and PARTE [Renaux, 2012] is created, that uses an actor-based approach to parallelize the algorithm. The actors are spread over multiple machines so their processing power requirements and memory usage is spread over the available machines.

Furthermore, a CED system has to take into account variations in load at run time. In a typical CED program, the events detected in the data stream will change during different 'phases' of the run time (e.g. a dancing game consists of an alternation between chorus and verses, or a traffic monitoring system alternates between periods with traffic jams and periods with speeding cars), which will have an effect on which parts of the program will use most processing power and memory at certain times. The system presented here contains a dynamic load balancer to redistribute the work at run time. This is achieved through *mobile actors*, a type of *code mobility*. Actors can move between machines at run time, i.e. they are mobile, in such a way that the work load is distributed evenly.

Contributions

In this dissertation, a rule-based CED system called DPARTE is developed. To our knowledge, DPARTE is the first Rete-based system designed for complex event detection that provides the following characteristics:

- The system runs on **multiple machines**, connected via a network, and the Rete nodes are distributed, resulting in a unified, distributed fact base of complex events.
- The system is **scalable**: as the amount of work increases (e.g. more sensors are used), it suffices to increase the number of machines to keep the performance intact.
- It is **elastic**: the amount of work can increase or decrease at run time, and the system adapts. New machines can be added to a running program and the work will be redistributed.
- It supports **dynamic load balancing**: as the type of work changes and different rules match, the work can be redistributed to use the available machines efficiently (the load is distributed evenly).

Overview of this thesis

The rest of this text is structured as follows:

Chapter 2: Large-scale pattern recognition gives an overview of the problems contained in large-scale pattern recognition. It discusses complex event detection, in particular rule-based pattern recognition, sketches the use case scenarios focused on in this thesis and their common properties, motivates the decision to use a distributed system, and concludes by specifying the requirements for the system developed in this thesis.

Chapter 3: Related work provides an overview of existing approaches to complex event detection and rule-based pattern matching, some of which are also parallel or distributed. It also presents code mobility and the properties exhibited by systems using code mobility.

Chapter 4: The Rete algorithm & PARTE describes in more detail the Rete algorithm and PARTE, upon which the system developed here is based.

Chapter 5: Distributing the Rete network details the changes made to PARTE to move it from a single machine to a network of machines with distributed memory. The architecture of the system and its specifics are outlined.

Chapter 6: Dynamic load balancing through mobile actors firstly presents mobile actors and how they are implemented, and secondly explains dynamic load balancing and how mobile actors are used for that purpose.

Chapter 7: Evaluation consists of a number of experiments that evaluate the system and check whether it meets the requirements set forth in chapter 2.

Chapter 8: Conclusion summarizes the contributions made in this thesis, sketches some possible directions of future research, and concludes the text.

Chapter 2
Large-scale pattern recognition

This chapter explores the problem domain of large-scale pattern recognition. We begin by describing complex event detection, followed by an overview of different approaches used to recognize patterns in data, after which the focus shifts to rule-based pattern recognition specifically. We describe a declarative language to define rules, and look at the general algorithm used to recognize patterns according to these rules. We list some use cases, and find that for large-scale pattern recognition, it is desirable to spread the computations over a distributed system. Lastly, a set of requirements is composed for the system developed in this thesis.

2.1 Complex event detection

2.1.1 Overview

Complex event detection (CED) – or also complex event processing (CEP) – is concerned with the detection of 'complex events' in streams of data.

Simple events are atomic events, typically detected by the hardware. For example: the current longitude and latitude as determined by a GPS device, the detection of an RFID tag, or the images captured by a camera. They contain the raw data as captured by the hardware.

Complex events are events that are composed of other (simple or complex) events. For example, whether a user is at home: this is not an event that is simply detected by the hardware, it is composed of simpler events such as the user's current location and the location of his home, and relationships between these events.

In complex event detection, simple events captured by the hardware enter the system, and the system tries to detect the complex events. The system can then react to

these complex events in an appropriate manner. Events have a significance, and one could say the aim of the CED system is "to make high-level sense out of low-level events" [Luckham, 2002] (Figure 2.1).

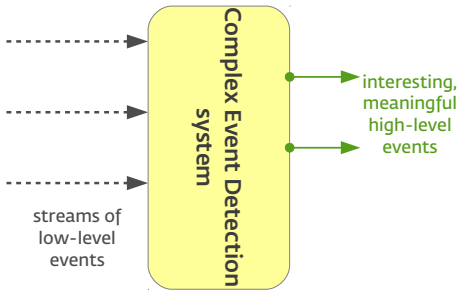


Figure 2.1: A CED system

2.1.2 Relations between events

Typically, the events captured by a CED system form streams of data. Some applications of CED include the detection of and reaction to the presence of RFID tags [Wang et al., 2006], algorithmic stock trading, and the monitoring of business processes such as fabrication lines. In these applications, a stream of data consisting of very low-level events is captured by hardware, e.g. a stream of detected RFID tags.

The aim of CED is to find the relations between these low-level events, find out what their significance is, and make sense of the data. We aim to detect meaningful events, so that we can react to them in an appropriate manner. The complex events we want to detect can be identified by detecting the relationships between lower-level events.

There are three common and important relationships between events [Luckham, 2002]:

- **Temporal relations**

  Events happen at specific times, and can be ordered according to this. It will often be important to know which events happened before or after another event. By attaching timestamps to events, their temporal relations can be compared.

- **Causal relations**

  One event can cause another event: event A *caused* event B if A had to happen in order for B to happen. Causality defines a dependency between events: if A causes B, then B *depends* on A.

Causality is an asymmetric property: if A caused B and A $\neq$ B, then B did not cause A. Two events are *independent* if neither caused the other to happen. Causality and time are related by the cause-time axiom [Luckham, 2002]: if event A caused event B, then no proper clock gives B an earlier timestamp than it gives A.

- **Aggregation**

  If the event A signifies an activity that consists of the activities of a set of events $B_1, B_2, ...$, then A is an aggregation of the events B_i. Aggregation is an abstraction relationship: event A is a higher-level event than B_i. It is a complex event, caused by the events B_i.

2.1.3 Event detection using pattern recognition

We want to pick up on particular events in streams of events, to quickly find the 'interesting' ones. We can do this by describing patterns of events. A **pattern** is a template that matches certain sets of events. It defines which sets you want to find – by describing the events in them and their relations.

There are several ways in which pattern recognition can be implemented.

A first, simple approach is what we call the **imperative approach**. Here, the programmer manually writes code to recognize the desired patterns in the data. This often results in many if statements, with nested conditions, testing the many variables captured by sensors.

There are many disadvantages to this approach [Hoste, 2010; Lü and Li, 2012]. First of all, it requires a lot of code that is written manually, which is a labor-intensive and error-prone process. To keep memory usage within reasonable limits, especially as more sensors are used, 'uninteresting' information should be garbage collected, but it is hard to determine which information is uninteresting. The resulting code is hard to maintain, and is hard to extend when new patterns should be recognized.

Another approach, perhaps the most frequently used, builds on **machine learning algorithms**, for example by Suma et al. [2011]. There are many machine learning algorithms used for pattern recognition, such as principle component analysis, hidden Markov models, neural networks, and support vector machines [Mitra and Acharya, 2007; Theodoridis and Koutroumbas, 2009]. A common property of all methods based on machine learning is that, to 'learn', these algorithms are trained using a large training set of already classified examples.

They do not suffer from the disadvantages of the previous approach, as the actual recognition of the patterns is not hard-coded but learned instead. Other advantages include that it becomes easy to extend the system with new patterns (just add them

to the training set[1]), and that frameworks built using these approaches are easily reused (they can just be retrained with another data set, although this training might take a lot of time).

However, there are also some disadvantages of this approach. First of all, it is necessary to have an – often large – set of classified examples to train the model. This set needs to be diverse enough so as not to accidentally train the model incorrectly, but in some scenarios (e.g. detecting a fire) there might not be many sample data for these situations. The classification of the training data also often has to be done manually.

Secondly, it is often quite hard (for the programmer) to see how the model has been trained. The training set is fed into the algorithm, and a trained model comes out, but this does not tell how it works internally: it is a black box. As a result, when data is misclassified, it is hard to find out how to fix this. This intransparancy is an often cited disadvantage of many machine learning methods [Benitez et al., 1997; Andrews et al., 1995].

Finally, the approach followed in this thesis is a **rule-based approach**: we define rules for every pattern, and data that match those rules are said to match that pattern. The rules are specified in a declarative way, using a domain specific language.

This approach does not suffer from the disadvantages of the previous two approaches. The declarative language used to specify rules makes it easy to define patterns and add new ones. It is also easy to see how the rules will react when certain data is fed into the system, and if a certain object is misclassified we can just look at the object and its contents, and see why it matches or does not match certain rules.

There are also some disadvantages to this approach: first of all an expert is needed to write down the rules. Furthermore, in some cases, it might be hard to write down the patterns in the form of rules. To remedy this, some approaches (such as Mudra [Hoste et al., 2011]) combine a rule-based approach with machine learning.

2.2 Rule-based pattern recognition

In this section, we define the rule-based language used for pattern recognition and describe the general algorithm used to match events to them.

[1] Machine learning algorithms can be extended by adding more samples to the training set, however many algorithms are not incremental, which means they need to be re-trained from scratch with the new, extended training set. This also implies one needs access to the original training set when extending it.

2.2.1 Language

The rules recognized by a rule-based pattern recognition algorithm can be defined using a domain-specific, declarative language. The language used in this thesis is the same as the language used by PARTE [Renaux, 2012] and Midas [Scholliers and Hoste, 2011], which is based on the languages used by CLIPS [Riley, 2013] and Jess [Friedman-Hill, 2013]. Some examples of the terms defined in this section are shown in Code listing 2.1.

Events are represented in the system as **facts**. A fact has a type (or template), and a list of slots, which are key-value pairs. A fact can be represented as an S-expression.

A **fact template** represents a type of fact. It has a unique name, and defines which slots a fact that uses this template can have. Each slot has a name and some options, such as its type, a default value, etc. (An example is given on line 2 in Code listing 2.1.)

A **pattern** is a partial description of a fact. It refers to a fact template, and can contain a list of slots. As it is a partial description, it does not need to contain all slots defined on its template. The values for the slots can be constants but also variables (these are prefixed with a '?').

A fact **matches** a pattern if it is of the same template, and every slot of the pattern containing a constant matches the corresponding slot of the fact. The variables in the pattern will be bound to the corresponding values in the fact. It is possible for the same variable to appear more than once in a pattern, if so, it is checked that all occurences of this variable bind to identical values.

A **rule** or **production** is made up of a left-hand side (LHS) and a right-hand side (RHS), separated by '=>'. The LHS can contain (1) patterns to be matched, and (2) tests to be executed on the variables bound by matching the patterns. The tests can use predicates such as '==', '<', 'and', 'or', and user-defined boolean functions. If facts are found that match all patterns on the LHS, and all tests succeed, the RHS is executed. The RHS contains **actions**, such as calls to user-defined functions, input and output, and assertions, retractions or modifications of new facts. In the RHS, it is possible to use the variables bound in the LHS (both when asserting new facts as for calls to user-defined functions). (Some examples are given in Code listing 2.1.)

The set of currenly asserted facts is called the **working memory**. Its contents, i.e. the facts that are currently considered true, are called the **working memory elements**.

2.2.2 General algorithm: the recognize-act cycle

Algorithms that match facts based on rules generally consist of three phases executed repeatedly, this is called the **recognize-act cycle** [Forgy, 1982; Gupta et al., 1986].

```
; Fact template
(deftemplate Point
    (slot x (type INTEGER))
    (slot y (type INTEGER))
    (slot name (type STRING)))

; Facts
(Point (x 1) (y 2) (name "point1"))
(Point (x 0) (y 0) (name "origin"))
(Point (x 8) (y 7) (name "point2"))

; Patterns
(Point (x 1) (y ?y) (name ?name))
(Point (x ?x) (y ?y) (name "point1"))
(Point (x ?a) (y ?a))
(Point)

; Rules
(defrule matching-y
    (Point (x ?x1) (y ?y) (name ?name1))
    (Point (x ?x2) (y ?y) (name ?name2))
=>
    (printout "Points " ?name1 " and " ?name2
        " have matching y coordinates."))

(defrule matching-y
    (Point (x 0) (y ?y1) (name ?name1))
    (Point (x ?x2) (y ?y2) (name ?name2))
    (test (> ?y2 ?y1))
=>
    (assert (Point (x ?x2) (y ?y1) (name "new"))))
```

Code listing 2.1: Some examples of fact templates, facts, patterns and rules.

Match phase During the match phase, the LHS of every rule is compared to the facts currently in the working memory. If the LHS of a rule can be satisfied, an **instantiation** of the rule is generated, this is a list of the working memory elements that made the match possible. This instantiation is added to the **conflict set**.

A simple algorithm for the match phase will consist of two nested loops. We iterate over every rule, and attempt to match the condition elements in its LHS with the elements currently in the working memory. Conceptually, this means a cross-product is made between the rules and the working memory elements. A naive algorithm therefore will scale very poorly as the amount of rules increases, the number of condition elements in rules increases, or the number of working elements increases. Gupta et al. [1986] show that in a typical production system, around 90% of the total execution time is spent on the match phase.

Conflict resolution phase In the conflict resolution phase, one of the productions in the conflict set is chosen for execution. If the conflict set is empty, execution halts.

The method used to choose a production from the conflict set is called the *conflict resolution strategy*. There exist a number of different strategies:

- Simply choose the first element from the conflict set.
- Give some rules explicit priorities (called "saliences") to prioritize them. (This is for example done by Jess [Friedman-Hill, 2013].)
- Some systems prefer more complex rules or patterns above simpler ones.
- Yet other systems analyze the results of rules to make an 'intelligent' choice.

In a parallel system, no such choice needs to be made: all matching productions can be executed in parallel.

Act phase Finally, in the act phase, the RHS of the chosen production is executed. During this phase, the system can do input/output, or change the working memory. New facts can be asserted, existing ones can be modified or removed.

Because this phase can change the working memory, the conflict set calculated in the match phase becomes invalid, so after this phase has finished we re-start at the first phase.

2.3 Complex event hierarchies applied to pattern recognition

Let us now look at some examples of how this rule-based pattern recognition language can be used in the context of complex event detection.

The first example is based on monitoring cameras placed around highways (Figure 2.2). These cameras can recognize cars in the image they capture, and can read the license plates of these cars using an optical character recognition (OCR) algorithm. This data can be used to gather information about speeding cars, traffic jams, etc.

Figure 2.2: Surveillance camera watching traffic.

For every camera, a fact of the template Camera is asserted. Every camera has a unique ID, and is on a certain highway at a certain position (km offset since 'origin' of highway) pointing in a certain direction. It also keeps track of the IDs of the previous and next cameras along the same highway.

```
(deftemplate Camera
  (slot id (type INTEGER))       ; Unique ID for the camera
  (slot highway (type STRING))   ; Name of the highway it's on
  (slot position (type FLOAT))   ; Position of camera along highway in km
  (slot direction (type STRING)) ; Direction in which the camera points
  (slot prev-camera (type INTEGER)) ; ID of previous camera along highway
  (slot next-camera (type INTEGER))) ; ID of next camera along highway
```

Whenever a camera sees a car, it asserts a fact of template CarInTraffic. The camera, through a computer vision algorithm, determines the license plate of the car, and the lane it is driving in. It also adds the time at which the car was seen.

```
(deftemplate CarInTraffic
  (slot camera (type INTEGER)) ; ID of camera that saw the car
  (slot plate (type STRING))   ; License plate of the car
  (slot lane (type INTEGER))   ; Lane it is on
  (slot time (type FLOAT)))    ; Timestamp at which camera saw the car
```

Imagine now we want to detect stolen cars on the roads. Whenever a stolen car gets reported, a fact of template StolenCar is asserted, containing the license plate of the stolen car.

```
(deftemplate StolenCar
  (slot plate (type STRING)))
```

We can then write a rule to detect a stolen car by trying to match cars seen by cameras with the same license plate as a stolen car:

```
(defrule StolenCarInTraffic
  (StolenCar (plate ?pl))
  (CarInTraffic (plate ?pl) (camera ?c))
  (Camera (id ?c) (highway ?hw) (direction ?d) (position ?p))
=>
  (printout "Sloten car " ?pl " seen on highway " ?hw
            " at position " ?p " going in direction " ?d))
```

The LHS of this rule will attempt to find three facts, with templates StolenCar, CarInTraffic and Camera, for which the plate of the StolenCar is equal to the plate of the CarInTraffic, and the camera slot of the CarInTraffic is equal to the id slot of the Camera. In other words, because the ?pl and ?c variables are repeated in different rules, their values are unified. The camera ID embedded in the CarInTraffic fact is used to find this camera, and get its highway, position and direction.

We can also detect speeding cars, by correlating the data collected by two cameras, as demonstrated in the next code snippet. When two cameras have detected the same car driving past them at different times, the average speed of the car can be calculated based on the distance between the cameras and these timestamps. If the speed is above 130 km/h, a message is printed.

```
(defrule CarTooFast
  (CarInTraffic (camera ?c1) (plate ?pl) (time ?t1))
  (CarInTraffic (camera ?c2) (plate ?pl) (time ?t2))
  (test (> ?t2 ?t1))
  (Camera (id ?c1) (highway ?hw) (direction ?d) (position ?p1))
  (Camera (id ?c2) (highway ?hw) (direction ?d) (position ?p2))
  (test (> (speed ?t1 ?p1 ?t2 ?p2) 130))
=>
  (printout "Car travelled too fast: " ?pl " at "
            (speed ?t1 ?p1 ?t2 ?p2) " km/h."))
```

This rule not only correlates attributes between different facts (the plates between the two CarInTraffic, and the camera IDs between a CarInTraffic and a Camera), but also does two extra tests on the captured variables. The first test is a simple check to prevent the rule from being triggered twice (if we would leave out this test, the rule would match once with one CarInTraffic matching the first pattern and another matching the second pattern, and once vice versa).

In the second test, a user-defined function 'speed' is called, which calculates the average speed of a car in km/h when it is seen on ?t1 at ?p1 and on ?t2 at ?p2, i.e. `(speed ?t1 ?p1 ?t2 ?p2) = (abs (* (/ (- ?p1 ?p2) (- ?t1 ?t2)) 3600))`.

Next, let us write a rule to detect stationary (non-moving) cars. Consider a car stationary if it has been within view of the same camera for more than 30 seconds.

When a stationary car is found, not only is a message printed, but a new fact of type StationaryCar is asserted as well.

```
(defrule StationaryCar
  (CarInTraffic (camera ?c) (plate ?pl) (time ?t1))
  (CarInTraffic (camera ?c) (plate ?pl) (time ?t2))
  (test (> (- ?t2 ?t1) 30))
  (Camera (id ?c) (highway ?hw) (direction ?d) (position ?p))
=>
  (assert (StationaryCar (camera ?c) (plate ?pl) (time ?t1)))
  (printout "Car not moving on highway " ?hw
            " at position " ?p " in direction " ?d))
```

This rule could be used to notify humans about a possible accident on the highway. Moreover, it can be used to recognize traffic jams. First, we recognize traffic jams within the view of a single camera: if there are three different stationary cars within the view of the same camera at the same time, the rule will match and a fact StationaryCars will be asserted, which contains the camera that saw the cars and the timestamp at which this happened.

```
(defrule StationaryCars
  (StationaryCar (camera ?c) (plate ?pl1) (time ?t1))
  (StationaryCar (camera ?c) (plate ?pl2) (time ?t2))
  (StationaryCar (camera ?c) (plate ?pl3) (time ?t3))
  (test (!= ?pl1 ?pl2 ?pl3))
  (test (< 0 (- ?t2 ?t1) 1))
  (test (< 0 (- ?t3 ?t2) 1))
  (Camera (id ?c) (highway ?hw) (direction ?d) (position ?p))
=>
  (assert (StationaryCars (camera ?c) (time ?t1)))
  (printout "Traffic jam on highway " ?hw
            " at position " ?p " in direction " ?d))
```

To recognize traffic jams, the next rule will correlate the information gathered by different cameras. If a camera (?c1) sees stationary cars, and the next camera along the highway (?c2) also sees stationary cars within a five minute window, the rule TrafficJam asserts that there is a traffic jam from ?c1 to ?c2. We continue to extend the traffic jam to the camera ?c3 following ?c2 with the rule TrafficJamContinued. This last rule will be called recursively for the length of the traffic jam.

```
(defrule TrafficJam
  (StationaryCars (camera ?c1) (time ?t1))
  (StationaryCars (camera ?c2) (time ?t2))
  (test (< 0 (- ?t2 ?t1) 300))
  (Camera (id ?c1) (next-camera ?c2))
=>
  (assert (TrafficJam (from-camera ?c1) (to-camera ?c2) (time ?t1))))

(defrule TrafficJamContinued
  (TrafficJam (from-camera ?c1) (to-camera ?c2) (time ?t1))
```

```
  (StationaryCars (camera ?c3) (time ?t3))
  (test (< 0 (- ?t3 ?t1) 300))
  (Camera (id ?c2) (next-camera ?c3))
=>
  (assert (TrafficJam (from-camera ?c1) (to-camera ?c3) (time ?t1))))
```

This examples demonstrates how simple, low-level events detected by the hardware (e.g. CarInTraffic) can be combined into evermore complex events (e.g. TrafficJam). We used a technique characteristic to Complex Event Detection: complex events are aggregations of simpler events, and events are stacked to form 'levels' of events. We have created an **event hierarchy**, in which Camera and CarInTraffic are at the bottom, StationaryCar combines them, StationaryCars forms the next level, and TrafficJam is at the top of the hierarchy (Figure 2.3).

Event hierarchies are a typical feature of CED systems. As another example, take a multi-touch interface: the simple events in this system are the positions of the fingers on the screen at a certain timestamp (detected by the hardware). A slightly higher level can relate events at different times to detect movements in e.g. a straight line or an arc. These could then be combined in a higher level that describes a sequence of such movements that form a shape, e.g. a square or a circle. In an even higher level, a sequence of such shapes could be detected.

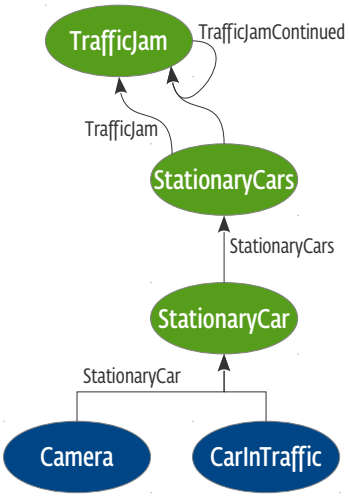


Figure 2.3: Complex event hierarchy for example of section 2.3. The blue events are 'simple events', as captured by the hardware; the green events are 'complex events', generated by rules. Arrows indicate which events are generated based on which other events, as defined by the labeled rules.

2.4 Use case scenarios

Next to the car monitoring example outlined above, there are some other use cases envisioned for the system described in this thesis. Amongst those are:

- Surveillance

  At large events, such as a festival, concert, football match or conference, cameras can be installed to monitor the crowd. They detect dangerous circumstances such as a fire or a panic outbreak, and notify human operators at a control center so that an appropriate reaction can be taken.

  A rule-based system can also be used to monitor the movement of crowds. *Crowd management* is concerned with the control of large crowds of people to ensure safety, e.g. by restricting access to an area after a certain number of people have entered it, by directing crowds along different routes, or by opening closed exits in case an evacuation is necessary [Berlonghi, 1995; Sime, 1995; Hu et al., 2004].

- Tracking

  The example in the previous section tracked cars, similar systems can be created to track and monitor other objects; for instance goods on an assembly line in a factory [Wang et al., 2009], the movement of endangered animals throughout their habitat, vital parameters (such as the heart rate and blood pressure) of a hospital patient, the concentration of fine particles and other pollution in the air, the water level of rivers and canals, the movement of packets over a network [Luckham, 2002], etc.

- Emergency management

  Rule-based systems can be applied to detect emergency and disaster scenarios, decrease the response time to such an event, and aid the mitigation and recovery. E.g. in Doherty and Rudol [2007], unmanned aerial vehicles (UAVs, such as small autonomous helicopters) are used in a search and rescue scenario. The UAVs have sensors installed on them, and can be used to locate victims.

  As an example, in case of a fire a set of UAVs with depth sensors and infra-red cameras mounted on them could be deployed. They fly through the burning building and aid firefighters in several ways. First, by detecting walls, doors and windows, they construct a map of the building on the fly, which firefighters can use to safely navigate through the building. Secondly, using an infra-red camera they can track the temperature throughout the building to find the origin of the fire and the rooms affected by it, so that firefighters know where to focus their efforts. Lastly, the UAVs can locate victims, and provide a safe path through the building for their evacuation.

Typically, in these scenarios there is not a large amount of training data available, while there exists expert knowledge. This makes a rule-based approach particularly suitable for emergency scenarios.

There are some common properties of these use cases that will be the focus for this thesis:

- They are **data-intensive**: patterns need to be detected among a huge amount of data. For example, the Microsoft Kinect camera can track a user's body, and generate a skeleton of 20 joints (20 three-dimensional coordinates) at a rate of 30 frames/sec. In our applications, we are interested in using multiple cameras.
- Data gathered by **multiple sensors** is **correlated**: we will specifically focus on scenarios where patterns stretch accross data captured by different devices. This increases the complexity of the problem, as data captured by different devices cannot be processed independently.
- We focus on **online processing**: data is processed as it enters the system. (As opposed to offline processing, where the data is gathered and then processed at a later time.) As such, the delay between capturing the data and recognizing the patterns in it should be minimal.

2.5 Motivation for distribution

As mentioned, in the use cases targeted by this thesis, there will be a focus on large-scale *data-intensive* applications, and applications where patterns are recognized by correlating data gathered by different sensors.

We envision that on the one hand the number of sensors will increase, and their resolution and frame rate will increase. On the other hand, we want to recognize ever more complex patterns in the data, and therefore more rules and more complex rules are needed. As a result, the required processing power and memory usage will increase.

The solution proposed in this thesis is to spread the problem over multiple machines. When the processing power or memory usage required for the specific application exceeds that of one machine, it should be possible to add another machine to the system and spread the processing power and memory requirements. Ideally, it should be possible to "just add more machines" whenever the complexity of the application increases. This property of a system is called *elasticity*.

Because the patterns that we want to recognize are not confined to one sensor, but instead stretch accross the data gathered by multiple sensors, it is not sufficient to just attach a processor to every sensor which processes the data of that sensor independently. Instead, all data needs to be gathered into one distributed system in which all data is present.

Furthermore, it is not possible for us to completely predict beforehand which parts of the system will use most memory. Therefore, we propose to use dynamic load balancing, through mobile actors. If at run time certain rules are triggered more, they (or parts of them) can move to a different machine.

This is especially useful in applications where rules are triggered in 'phases'. For example, in the traffic monitoring application above, there will be periods during which there are many traffic jams – so the rules matching traffic jams are very active – while during other periods other rules there are no traffic jams but other rules are active – for instance those matching speeding cars. Similarly, in an application that recognizes dance moves, there will be periods during which a certain subset of moves are recognized (e.g. the chorus of a song) while at other times other moves are more frequent (the verses of the song). Or, when monitoring crowds, these crowds move and will at one moment be seen by one camera and at another by another camera.

Our system should provide a mechanism to cope with these changing patterns in processor and memory usage, and adapt. We will do this through *dynamic load balancing*, using mobile actors.

2.6 Requirements

Based on the problem presented in the introduction and this chapter, we can now make a list of requirements for the system of this thesis:

Optimized for rule-based CED The system should be designed for Complex Event Detection, using rules for pattern recognition. It should focus on systems that correlate events from different sensors.

Online processing It should be suitable for online processing, i.e. data is processed as it enters the system. The delay between the hardware capturing the data and the pattern being recognized should be small enough, in the range of a few seconds maximally.

Scalability The system should scale over multiple machines. For instance if the amount of sensors increases, the amount of data that enters the system will increase, and by adding extra machines to the system it should be able to handle this increase in data. The throughput of the system should increase as the amount of machines increases.

Elasticity Even as the amount of work varies at run time, it should be possible for the system to deal with this by allowing additional machines to be added. The work should then be redistributed to make efficient use of the available machines.

Dynamic load balancing As the data that enters the system changes, and the processing power requirements and memory usage change between the machines, the load should be balanced between the machines automatically. The division of the work over the machines should be able to change at run time, dynamically.

Chapter 3 examines related work in the context of these requirements. Chapter 4 examines the Rete algorithm, and PARTE, in more detail.

In Chapter 5, a system called DPARTE is constructed, based on PARTE, that is optimized for rule-based CED and online processing, and scales over multiple machines. In Chapter 6, this system is extended to provide elasticity and dynamic load balancing. This system is then evaluated in Chapter 7, on the basis of these requirements.

Chapter 3
Related work

This chapter gives an overview of related work. It focuses on three areas of research: (1) complex event detection, i.e. detection of 'complex events' in streams of data, (2) rule-based pattern matching, i.e. detection of patterns in data using rules, and (3) mobile actors, i.e. actors that can 'move' between different computational environments at run time.

3.1 Complex event detection

Complex event detection (CED) is concerned with the detection of *complex events* in streams of data, as explained section 2.1 in the previous chapter. A lot of research in CED systems has focused on detecting RFID tags and processing streams of such data [Wu et al., 2006; Wang et al., 2006]. Languages have been developed for rule-based CED (e.g. SASE [Wu et al., 2006] and RAPIDE [Luckham, 2002]) – with support for temporal operators and the ability to define complex events composed of other events – which can also applied in the context of this thesis.

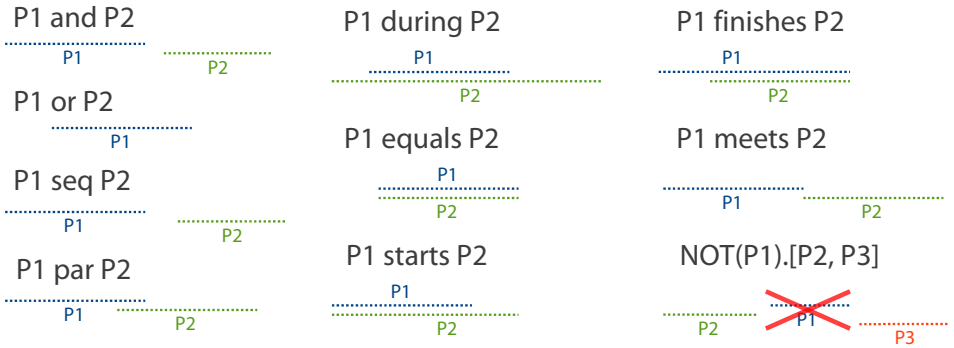


Figure 3.1: Temporal operators proposed by Anicic et al. [2010]

For instance, the language proposed by Anicic et al. [2010] contains the following temporal operators between events (illustrated in Figure 3.1):

- P1 AND P2: P1 and P2 happen, it does not matter in which order.
- P1 OR P2: P1 and/or P2 happen.
- P1 SEQ P2: P1 is followed by P2, i.e. P2 starts somewhere after P1 ends.
- P1 PAR P2: P1 and P2 happen in parallel: they overlap for a non-zero time.
- P1 DURING P2: P1 happens during P2, i.e. P1 starts and ends during P2.
- P1 EQUALS P2: P1 and P2 happen during the exactly same time interval.
- P1 STARTS P2: P1 starts at the same time as P2, but ends earlier.
- P1 FINISHES P2: P1 ends at the same time as P2, but starts earlier.
- P1 MEETS P2: P1 ends at the exact time that P2 starts.
- NOT(P1).[P2,P3]: P1 does *not* happen between the end of P2 and the start of P3.

The system developed in this dissertation builds on PARTE [Renaux, 2012], and the set of operators defined by PARTE is retained. As in PARTE, in our system as well it is possible for the user to define custom operators.

3.2 Rule-based pattern matching

This section will focus on algorithms used for rule-based pattern matching, and the parallelization of these algorithms.

Expert systems are systems that encode 'knowledge' (or 'expertise') about a certain task [Liao, 2005]. This knowledge is stored in the computer, and users can call upon it when needed. The computer can make interferences and arrive at specific conclusions. There are many different categories of expert systems, including rule-based systems, knowledge-based systems, neural networks, fuzzy expert systems, intelligent agent systems, etc.

We focus specifically on rule-based systems, known as *production systems*. A program for a production system consists of facts, and rules that describe relations between these facts (as described in section 2.2). The production rules encode heuristic 'knowledge' on the problem domain, in the form of if-then rules. The rules can be used to recognize patterns in data.

3.2.1 Expert systems

Backward and forward chaining

There are two approaches to rule-based systems. One is *backward chaining*, where the system tries to prove a query (the goal) by working 'backwards' from the goal,

using the rules, to find whether the necessary facts are present so that the query can be satisfied. In other words, the consequent of the rules will be matched through the current fact base, and in case of a match the antecedent of the rule is added to the fact base. Prolog is an example of a programming language that uses backward chaining, widely used to implement rule-based expert systems [Clocksin and Mellish, 2003].

The other approach is *forward chaining*, in which facts travel 'forward' through the rules, if they match the antecedent of a rule the consequent is added to the fact base.

You could consider backward chaining a 'query-driven' approach (from the query/-consequent to the antecedent), and forward chaining 'data-driven' (from the data/antecedent to the consequent). In this thesis, a data-driven approach is more appropriate as data should be processed while it is detected by sensors, to recognize patterns as soon as they appear. In a backward chaining approach it would be necessary to continually query the system for patterns when they are needed, instead a forward chaining approach notices a pattern as soon as it is detected.

Rete

The Rete algorithm [Forgy, 1982] is a pattern matching algorithm in which a large collection of facts is compared to a large collection of patterns, and the set of matches is determined.

The basic idea behind the Rete algorithm is to compile the rules into a network. When a fact is asserted, it will travel through this network. The nodes through which it passes will perform tests on the components of the fact, that correspond to the tests on the LHS of the rules. When a fact reaches a terminal node, it is known to match a rule.

The Rete algorithm is a particularly fitting algorithm for our problem domain. First of all, it is forward chaining, and new facts can continually be asserted as they are detected by the hardware. These new facts can be combined with old facts to match new patterns.

Secondly, the Rete algorithm achieves a speed-up by avoiding iteration over rules and facts: iteration over rules is avoided by compiling them into a network, and iteration over facts is avoided by saving state in the nodes of the network. This efficiency is necessary in systems with a lot of rules and facts, like in our use case. This saving of state is only useful in case the fact base is relatively stable: the algorithm saves state between two cycles of the recognize–act algorithm, so saving state is only useful if most state does not change between two cycles. The condition for a stable fact base is met in our envisioned use cases: every rule will contain at most some assertions in its RHS, so after a rule has been executed only a few facts are added to the fact base. Saving state is a trade-off of memory for speed.

Lastly, Rete avoids checking common subparts of different rules multiple times. This is useful in our use case scenarios, as many rules may feature similar patterns. It

improves the efficiency of matching complex rules.

There are several popular rule engines that use the Rete algorithm, such as CLIPS [Riley, 2013], Jess [Friedman-Hill, 2013] and Drools [JBoss, 2013].

TREAT

TREAT [Miranker, 1987; Miranker and Lofaso, 1991] is an alternative algorithm to Rete. It builds on the conjecture of McDermott et al. [1977] that the cost of maintaining the state of condition elements between cycles is more expensive that the cost of recomputing these comparisons. This is based on the observation that the memory used by the Rete beta-memories can be combinatorially explosive, in which case the system will have to rely on virtual memory (swap). This will slow down the system, which is why recalculating is faster than saving.

Additionally, Miranker notes that, on a shared-memory parallel computer (such as originally proposed by Forgy), Rete tokens can be manipulated by simple writes to memory; but on a distributed-memory parallel computer (in his proposal) manipulating tokens can involve costly communication. Here too, it is preferable to recalculate information rather than saving and communicating it.

However, in our case, because of our focus on online processing (section 2.6), recalculating information is not desirable: this will lead to recalculating the matches for entire patterns when only one of its condition elements has changed.

One other interesting observation by Miranker is that, because the "join" operation in Rete in commutative and associative, it is possible to re-arrange the order in which these joins happen. Three possible orderings were studied:

- Static ordering: joins happen in the same order as the condition elements are written in the rule. This is the approach taken by Rete.
- Seed ordering: alpha memory that changed during this iteration of the recognize–act cycle is considered first.
- An ordering based on semi-join reductions: this approach was deemed unsuccessful and is not expanded upon.

LEAPS

LEAPS [Batory, 1994] is another algorithm for production systems. LEAPS translates programs into C code, and relies on carefully selected search algorithms and data structures to speed up the rule matching.

Whereas Rete and TREAT materialize all tuples that satisfy the predicate of a rule, LEAPS avoids this by doing lazy evaluation of tuples, that is, they are only materialized when needed. This reduces the space and time complexity of LEAPS.

When a fact is asserted, it receives a timestamp and is put on a stack. The elements in the stack will be sorted by their timestamp, the most recently updated element is at the top. During a cycle, the fact at the top of the stack is selected, and is used to 'seed' the selection predicates of all rules (these selection predicates are the patterns on the LHS of the rule). Rules are seeded in a particular order to increase efficiency: the rules with most positive (i.e. non-negated) condition elements will be seeded first. If the selected fact cannot seed a given rule, we move on to the next rule. If it can seed a rule, the rule is fired.

However, LEAPS is specifically optimized to produce fast sequential executions, and is therefore difficult to parallelize. Its lazy evaluation is also not useful in a system such as ours, where all matching productions are fired. As such, it is not applicable for the use cases envisioned in this thesis.

Soar

Soar [Laird, 1987] is an architecture for an "intelligent system capable of solving tasks". It also uses a production system to represent its knowledge: when elements in the working memory match the conditions of its rules, the rules are fired. Internally, it uses the Rete algorithm for pattern matching.

Furthermore, Soar has a learning mechanism that relies on *chunking*: it creates new productions, called 'chunks', that summarize the results of previous problem solving, and these are added to the set of existing productions. These chunks can then fire later.

As chunks are added during the excution of the program, the system becomes slower. Soar/PSM-E [Tambe et al., 1988] is an implementation of Soar that tries to optimize Soar by using parallelism. In this implementation, Tambe et al. examine several techniques aimed to improve parallel performance for Soar.

One of these techniques consists of converting the long chains of join nodes produced by Rete for rules with many condition elements into a 'constrained bilinear network'. To form these networks, condition elements are grouped and matching can happen for different groups in parallel. Figure 3.2 shows an example. Such a technique, although not currently applied in our system, can be useful for us too: the different branches of the constrained bilinear network can be spread over multiple machines.

Conclusion

Based on this overview of different algorithms of expert systems, it seems that the Rete algorithm is most fitting for our system. First of all, it fits our use case scenarios: it is forward chaining, efficient, and it avoids checking common subparts of different rules multiple times, which is a situation likely to arise when matching complex rules.

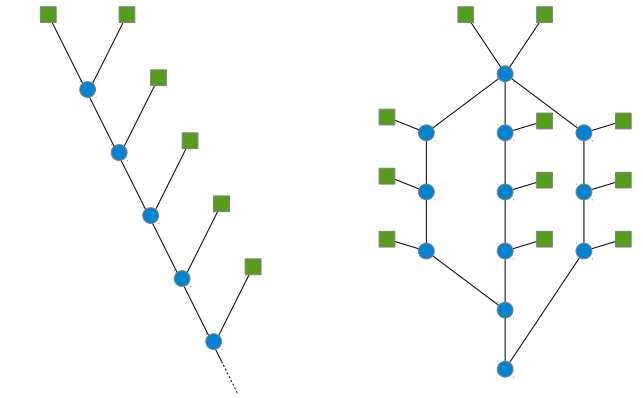


Figure 3.2: Rete networks that match rules with many condition elements (11 in this example). Alpha memories are shown as green squares, join nodes as blue circles. On the left, a traditional Rete network for such a rule is shown: the 11 CEs are joined by a sequence of 10 join nodes (only 5 are drawn). On the right, a constrained bilinear network that joins 11 CEs using 12 join nodes. In the second case, there is more possibility for parallelism.

Furthermore, it is parallelizable and can be distributed over multiple machines, because the rules are converted into a network of nodes. Each of these nodes can work separately, and they can be spread over multiple machines. Dynamic load balancing is possible by moving nodes between machines.

3.2.2 Parallelization and distribution of the Rete algorithm

There are several articles discussing the parallelization of the Rete algorithm for multiprocessor systems and multiple computers. This section briefly reviews them and discusses the ideas suitable for our distributed system.

Gupta et al.

Gupta et al. [1986] describe a parallelization of Rete 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. Every node of the Rete network is considered a 'task'.

Two limitations apply to the nodes: "(1) each node is permitted to process only one input token at a given time, and (2) the node has only a single thread of control internally." This in effect means the nodes become actors as in the actor model proposed by Agha [1985]. They also state that both these restrictions can be relaxed in

certain cases.

This type of parallelism is fine grained, as individual *tokens* are processed in parallel: an average task takes only 50–100 machine instructions. This fine granularity means that special care should be taken to avoid possible overheads, e.g. during the scheduling of the tasks. Because access to a large amount of shared data is needed, this approach is suitable for multiprocessor (or multi-core) systems with shared memory, but not necessarily for systems with distributed memory.

They report an average true speed-up of 8.25 (when using 32 processors, compared to the best uniprocessor system), citing that the reasons for the small speed-up are:

- The number of changes to the working memory per recognize–act cycle is small.

  When a working memory element is changed between cycles (newly asserted, modified or retracted facts), the rules relevant to it must be re-evaluated. As more working memory elements change, there are more rules that have to be re-evaluated, so more parallelism can be exploited. However, in most systems the number of changes per cycle tends to be small: generally less than 0.5%.

- The number of rules relevant to each change to the working memory is small.

  Not only are just a few working memory elements changed between cycles, but they also affect few rules. If there are more rules affected by the change, there is a larger opportunity for parallelism. However, in general the number of rules affected by a change in the working memory is quite small compared to the total number of rules.

- There is a large variation in the processing required for the relevant rules.

  Generally, most rules only require a small amount of processing, while a few require much more. These latter determine the amount of exploitable parallelism. If there are few such rules, the amount of parallelism is limited.

The system presented in this thesis builds on PARTE (see later), which uses the approach of Gupta et al. This solution was chosen because it builds on the actor model, which can be distributed over multiple machines by spreading the actors over the machines. Dynamic load balancing can be added by moving actors between machines.

Because each condition element in a rule becomes a separate node in the Rete network, and therefore a separate task in the approach by Gupta et al, complex rules will be parallelized and if needed spread over multiple machines. In our use cases, there might be many complex rules, and it is necessary that these can be parallelized and distributed.

Kelly and Seviora

Kelly and Seviora [1987] describe an architecture for parallelizing Rete on a multi-

processor machine. In this approach, a partitioning even smaller than the one used by Gupta et al. is used: instead of partitioning at the level of nodes, this approach partitions at the level of token–token comparisons.

In this architecture, 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 process. These copies can now be distributed over a large number of processors.

This architecture offers a high degree of parallelism, but introduces more communication. It is therefore less suitable for this thesis, as communication will be more expensive in a distributed system. In our case a coarser granularity of parallelism is more suitable.

In small-scale simulations ran by Kelly and Saviora, the speed-up achieved by this approach is 4. They estimate that in typical production systems, their approach reaches a speed-up in the order of 300.

Lana-Match

Lana-Match [Aref and Tayyib, 1998] is an approach to distributing Rete over multiple machines. This algorithm uses an architecture in which there is one Controller Processor (CP), and several Slave Processors (SP). Each of these has a local copy of the Rete network. The CP will send activations to an SP, which executes them. The SP keeps a list of facts that are added to and removed from the fact list. When the SP finishes, it sends back this list to the Controller.

All SPs will be executing activations in parallel. To prevent inconsistencies, Lana-Match relies on a centralized timestamp optimistic concurrency model. When the CP sends an activation to an SP, it attaches a timestamp (an incrementing integer). When the SP finishes and sends the list of facts to add/remove back the to CP, the CP does not commit these changes immediately, instead, it adds this list to a queue (the Action Queue).

When the CP has free time, it examines the Action Queue. It will commit the changes in the queue, but based on the timestamp (from old to young, so in the same order as if the activations were done sequentially). If it comes across an activation that has not finished yet, it waits for it to finish before committing any other changes.

The authors describe the architecture of the system, but did not implement it. Therefore, there are no measurements of achieved speed-up.

The advantages of this model, as described by Aref and Tayyib, are:

- The model is adaptive (scalable): adding a SP is very easy.
- Run time parallelism: this model exploits parallelism at run time, instead of relying on compile time analysis.

- Reliability: if an SP goes down, this is no problem. (The CP cannot go down however.)
- The programmer (of the rules) does not need to care about parallelism.

They list as disadvantages that, firstly the CP can be memory intensive (because all changes are buffered in the action queue before they are committed), and secondly the CP can become a bottleneck: in a large system, the CP might not be able to keep up with the SPs. They suggest to tackle these issues by using a more powerful system for the CP than for the SPs.

However, in our use cases these disadvantages might become more severe: because we focus on large-scale pattern recognition, in which data-intensive streams are processed, the action queue of the CP can become a bottleneck that cannot be circumvented by using a more powerful system.

Another disadvantage is that, because Lana-Match relies on a centralized optimistic concurrency model, it assumes each rule will only change a very small part of the fact base, and that conflicts are rare.

PARTE

PARTE [Renaux, 2012] is a parallelization of the Rete algorithm based on the approach of Gupta et al. [1986]. It uses the actor model [Agha, 1985], as every node of the Rete network becomes an actor. Different actors can run simultaneously, but each actor internally only has one thread of control. As in the approach of Gupta et al., multiple tokens will be flowing through the Rete network simultaneously.

PARTE is specifically designed for the detection of user-interface patterns such as gestures captured by a camera. To this end, it provides soft real-time guarantees by providing predictable upper bounds on the execution time which it requires to recognize patterns.

Another particular feature of PARTE is that it can automatically remove expired facts, which is useful when working with data that contains temporal information (which is the case in Complex Event Detection).

Several experiments show that PARTE achieves a good speed-up when parallelized on commodity multi-core machines.

PARTE seems to be the best fit amongst the algorithms presented in this section for the use case in this thesis. It has similar requirements, such as the requirement for online processing, and a similar focus on event-based CED systems.

Parallel rule firing

Ishida [1991] describes an alternative parallel execution model for rule engines. In this approach, multiple rules will be *fired* – not matched – in parallel: the act phase

of the recognize–act cycle (see subsection 2.2.2) is parallelized (i.e. the RHS of rules), not the match phase (the LHS).

This system analyzes which conditions need to be satisfied for the parallel result to be the same as the sequential result, by constructing a data dependency graph and using it to detect interference. A speed-up of 2–9 is reported.

Ishida lists three sources of parallelism in production systems:

- Rule parallelism: multiple rules are fired in parallel. No communication between is needed. More parallelism is possible as the amount of *independent* rules increases.
- Pipeline parallelism: multiple rules are fired in parallel, passing data from one to another in a pipeline. More parallelism is possible as the length of the pipe increases.
- Data parallelism: the same rule is fired multiple times, for distinct data. As the number of elements in working memory increases, more parallelism is possible.

The approach followed by Ishida is not specific to Rete or any of its variants, as these algorithms focus on improving the match phase while this approach focuses on the act phase. It could therefore be combined with any of the previous approaches.

In the system presented in this thesis, multiple rules will be fired in parallel: the RHS of each rule is a terminal node in the Rete network and can run independently. In other words, our system uses rule parallelism.

3.3 Mobile actors

The approach followed in this thesis uses mobile actors. Mobile actors are actors that can move between computational environments. They constitute a type of code mobility. **Code mobility** is defined as the capability to dynamically change the bindings between code fragments and the location where they are executed [Fuggetta et al., 1998].

3.3.1 Actors

The basis for the actor model was originally outlined by Agha [1985]. The actor model is a model of concurrent computation, in which actors are the primitive units of parallelism.

An **actor** is a computational entity that, in response to a received message, can:

- Send messages to other actors.
- Create new actors.

- Modify its internal state.

It is impossible for an actor to view or modify the internal state of another actor. This prevents race conditions. All communication between actors should happen through messages: actors can send asynchronous messages to each other.

Actors internally have a single thread of control. This means no two messages can be processed simultaneously by the same actor, this is a feature that prevents race conditions. It is however possible for different actors to process messages at the same time, for example on different cores or on different machines. This is what enables the parallelization of programs using the actor model.

Messages can be sent between actors by using an actor's **address**. An actor can only communicate with actors whose address it has, which it can obtain from messages it receives or if it is an address of an actor it has itself created.

When a message is sent to an actor, it is put in the **inbox** of that actor. This inbox is a queue, when an actor is scheduled to execute it will pop messages from this queue to process them. The inbox should guarantee that no race conditions can happen to it, e.g. by guarding the queue by a lock or by using a lock-free data structure for the queue. There is no guarantee on the order of the delivery of messages.

3.3.2 Mobile actors

Mobile actors or mobile agents are software components that can move be tween different computational environments [Lange et al., 1997]. To transform an actor into a "mobile" actor, we give it the ability to "move" at run time.

Before this process of moving, the computations running on the actor are "paused". The actor and its internal state are serialized, and move to another machine. There, they are deserialized and the computation is "resumed". In other words, three elements of the actor move:

1. its **code**, i.e. the instructions being executed;
2. its **internal state**, these are the values of its internal data structures; and
3. its **execution state**, i.e. at what point in the computation the actor was paused.

Some of the unique characteristics of mobile actors are [Lange et al., 1997]:

- Autonomy: a mobile actor is autonomous, i.e. it contains sufficient information so that it can function on its own, and make decisions on its own.
- Concurrent execution: multiple mobile actors can be executed simultaneously.
- Local and remote interaction, via asynchronous message passing: mobile actors can interact with other actors both on the same machine and on a different machine, they do this by passing messages asynchronously.

The mobile actor paradigm differs from other mobile code paradigms in the fact that an existing computational component moves.

3.3.3 Application to this system

In the approach of this thesis, nodes of the Rete network become mobile actors, and move between machines to provide dynamic load balancing. Following the terminology established by Fuggetta et al. [1998], the machines in our system are the *Computational Environments* (CEs) and the moving nodes are the *Executing Units* (EUs).

In more traditional systems supporting code migration, this migration is often transparent: the programmer has neither control nor visibility of the process migration. However, systems supporting code mobility take another approach: location is a pervasive concept that is not hidden from the programmer, he is aware of it and can use it, and can control the mobility. However, our approach is closer to the traditional one: the rule programmer should not worry about where his code runs, the system should take care of that for him.

The system of this thesis can be compared with a "Distributed Information Retrieval" system, considered by Fuggetta et al. to be the "killer application" for the mobile actor paradigm. In this application, information is dispersed throughout a network, and the code moves to be 'closer' to the data it uses. In our system too, information is dispersed throughout the network, although we control how the information moves. While the original intent behind Distributed Information Retrieval systems is to move code to the information it needs, because it might be impossible or unfeasible to move the information (e.g. because of its huge size), this is not applicable in our use case. Instead, in our system, the code moves to the computational resources it needs.

3.3.4 Properties of systems with code mobility

Fuggetta et al. define the concepts of *strong* and *weak mobility*, strong mobility is the ability of a system to allow migration of both the code and the execution state of an EU to a different CE; while weak mobility only allows the migration of code and some initialization data, but not the execution state. For our use cases, it is desirable to have strong mobility: an actor moves to another machine, and resumes execution there from the exact point at which it was suspended.

Within strong mobility, there are two mechanisms: *migration* and *remote cloning*. In the migration mechanism, an EU is suspended, transmitted to the destination CE, and resumed there. In remote cloning, a clone is made of the original EU at the destination CE, but it is not destroyed at the origin CE. For our system migration is required.

Migration can be either *proactive* or *reactive*. In a proactive migration, the time and destination of the migration are determined autonomously by the migrating EU. In a reactive migration, the migration is triggered by an external EU, e.g. an EU acting as 'manager'. The system implemented in this thesis will use a reactive mechanism: a centralized 'master' determines when and where migrations should happen; although a proactive mechanism would work as well.

3.4 Summary

This chapter provided an overview of Complex Event Detection, rule-based pattern matching, and mobile actors.

After an overview of different rule-based pattern matching algorithms, it was decided that the Rete algorithm [Forgy, 1982] is the best fit for our problem domain, as it is forward chaining, efficient, and can be distributed over multiple machines.

Looking at different ways to parallelize and distribute Rete, it was concluded that the approach taken by Gupta et al. [1986] and extended upon by PARTE [Renaux, 2012] is a good fit for this thesis: it uses the actor model and can therefore be distributed over multiple machines. By making every node of the Rete network a separate actor, and because Rete converts complex rules into many nodes, this approach will allow to spread complex rules over multiple machines.

Lastly, an overview was given of the actor model [Agha, 1985] and the mobile actor paradigm [Lange et al., 1997], which will be applied to add dynamic load balancing to our system. The actors will move between machines at run time based on information gathered by a load balancer, i.e. a reactive migration.

Chapter 4
The Rete algorithm & PARTE

The system introduced in this thesis implements a distributed version of the Rete algorithm, which is described in the first half of this chapter. It builds on PARTE, a parallel (multi-core) implementation of Rete, PARTE is described in the second half of this chapter.

4.1 The Rete algorithm

Rete is an efficient pattern matching algorithm for rule systems, first described by Forgy [1982].

The basic idea behind the algorithm is to compile the declared rules into a network, called the Rete network. This is also where the algorithm got its name, "rete" is Latin for "network". When facts are asserted, they will travel through the network, and reach terminal nodes when they match a rule.

The Rete algorithm achieves a speed-up by avoiding iteration over rules and facts during the match phase of the recognize-act cycle. The iteration over rules is avoided because they are compiled into a network, the iteration over facts is avoided because the nodes in the network maintain some state. This saving of state is a trade-off between memory and speed.

4.1.1 Tokens

Working memory elements travel through the network in *tokens*. A token consists of a tag and a list of key-value attribute pairs. The *tag* is either +, meaning that the accompanying attributes represent a fact added to the working memory; or −, meaning that they represent a fact retracted from the working memory.

So, when the following fact is asserted:

```
(Point (x 1) (y 2) (z 3))
```

The following token will be sent into the network:

```
<+, (Point (x 1) (y 2) (z 3))>
```

When an element in the working memory is modified, this can be represented by a retraction (−) of the old element followed by an assertion (+) of the new element.

Modifying:

```
(Point (x 1) (y 2) (z 3))
```

into:

```
(Point (x 1) (y 2) (z 4))
```

will cause the following two tokens to be sent into the network:

```
<-, (Point (x 1) (y 2) (z 3))>
<+, (Point (x 1) (y 2) (z 4))>
```

4.1.2 Elements to test

The declared rules are compiled into a network of nodes. The nodes test the features of the tokens. These features are grouped into two classes: intra-element and inter-element features. Additionally, our system supports beta tests to test relations between captured variables.

Intra-element features are features that concern only one working memory element. For example, for the pattern:

```
(Point (x ?x) (y 8) (z ?x) (name ?n))
```

The following intra-element features should be tested:

- The class of the fact should be 'Point'.
- The value of its y attribute should be 8.
- The values of its x and z attributes should be equal.

On the other hand, **inter-element features** concern features that occur in multiple patterns. For example, consider the rule:

```
(defrule matching-y
    (Point (x 1) (y ?y))
    (Point (x 2) (y ?y))
=>
    [...])
```

There are two lists of intra-element features to check for each of the two patterns (to check the class and x attribute of the facts), and an additional test to check that the y attributes of both facts match: an inter-element feature.

Additionally, our system supports **beta tests**, these are tests on the variables captured by the patterns in the rule. For example:

```
(defrule matching-x-and-larger-y
    (Point (x ?x) (y ?y1))
    (Point (x ?x) (y ?y2))
    (test (> ?y1 ?y2))
=>
    [...])
```

Tests are encapsulated in an S-expression tagged 'test', and evaluate to a boolean value. A test consists of a predicate and a list of arguments, which can be literals (e.g. a number or a string), variables captured in previous patterns, function calls (e.g. an addition) or another test. For example: `(test (and (> ?x1 ?x2) (== (+ ?x1 ?x3) 8)))` tests whether ?x1 is larger than ?x2, and ?x1 + ?x2 is equal to 8.

Different implementations of the Rete algorithm support different predicates and functions, but generally arithmetic operators (+, -, *, /), comparison operators (==, <, >, <=, etc.) and boolean operators (and, or, not) are included. Our system, like CLIPS and PARTE, allows the user to define his own predicates and functions. For instance, such a function could calculate the Euclidean distance between two points, check whether a certain point is within a bounding box, or any other desired problem-specific functionality.

4.1.3 Compiling the rules into a network

Compilating the rules will build a network, by linking together nodes that test for the features described in the previous section. When a LHS is encountered, the compiler will start by building nodes to test the intra-element features, these are the **constant-test nodes**. It creates a node for every such feature for every pattern in the rule, and links them together in a linear sequence.

Next, it will create nodes to check for the inter-element features. These nodes have two inputs and check a feature between the two patterns arriving at its inputs, they are the **two-input nodes** (also called **join nodes**). The first node will join the ends of the linear sequences testing the intra-element features of the first two patterns in the rule, the following node joins the output of this one with the end of the linear sequence for the third pattern, and so on.

Whenever a beta test is encountered in the rule, a **beta-test node** is inserted which will perform the defined test.

After the last two-input node or beta-test node of the LHS of the rule, the compiler inserts a special **terminal node**, which represents the RHS of the rule. When a to-

ken reaches this node, the rule has been successfully matched and the RHS can be executed.

Figure 4.1 contains a diagram of the network for the 'matching-y' rule shown above.

Figure 4.2 is a diagram of the Rete network for the traffic monitoring example of section 2.3.

4.1.4 Types of nodes

Let's take a closer look at the types of nodes and their inner workings.

Root node Not mentioned in the previous paragraph was the root node. Every Rete network starts at the root node. Asserted facts arrive here and are sent to the constant-test nodes at the start of the linear sequences described above.

Constant-test nodes As described in the previous paragraph, constant-test nodes test the intra-element features of tokens. Since these nodes are at the start of the network, every token they receive is a simple fact, with a class and some attributes set to constant values.

There are three types of intra-element features that can be tested:

- The class of a fact
- The equality of one attribute to a constant
- The equality of two attributes

A useful optimization to the Rete algorithm consists of removing the root node and the constant-test nodes that check for the fact class. Because a fact can only belong to one class, an asserted fact will be sent to root node, which passes it to every class-checking constant-test node, and only one of them will forward it to its successors while the others discard it. With this optimization, the token is sent to the successors of the matching class-checking constant-test node immediately. The root node and the class-checking constant-test nodes become redundant.

Two-input nodes A two-input node performs a conjunction on the tokens arriving at its two inputs. Two-input nodes are *state-saving*: they contain two lists called the left and right memory. The left memory contains a list of copies of tokens that arrived at its left input, the right memory of tokens that arrived at its right input.

When a token arrives at the left input, this node will try to match it with all of the tokens in its right memory. For every match, a token is sent to the successors of this node. The tag of the sent token is the same as the tag of the token that just arrived at the left input. The process for a token that arrives at the right input is similar.

```
1        (defrule matching-y
2            (Point (x 1) (y ?y))
3            (Point (x 2) (y ?y))
4        =>
5            [...])
```

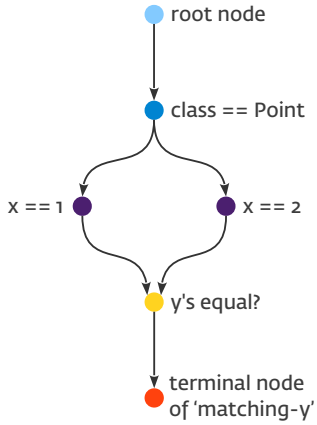


Figure 4.1: A simple rule matching two points, and the corresponding Rete network.

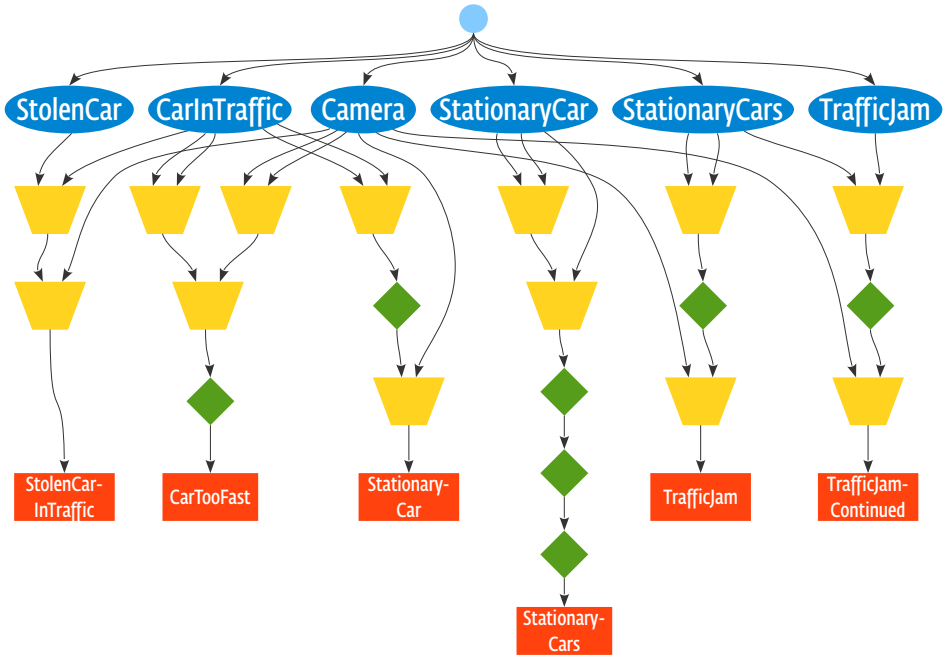


Figure 4.2: The Rete network for the traffic monitoring example of section 2.3. The light blue circle is the root node, where facts enter the system. The dark blue ellipses are constant-test nodes, here they test the class of the incoming tokens. The yellow trapezoids are join nodes, and test the inter-element features of the incoming tokens. The green diamonds are beta-test nodes, one for every `(test ...)`. The orange-red rectangles are the terminal nodes, one for every rule: they execute the RHS of the rule.

When a token with a + tag arrives, it is stored in the memory of the input at which it arrived. When a token with a − tag arrives, any token with an identical data part is removed from the memories.

Two-input nodes for negated patterns A special type of two-input node is needed for negated patterns, e.g:

```
-(Point (x 2))
```

which matches all facts excepts Points with an x attribute equal to two.

This node stores a counter along with each token in its left memory. The counter indicates the number of tokens in the right memory that match this token. In the right memory the tokens that match the negated pattern are stored (for the example above, all points with x attribute 2). Tokens with a counter of zero can pass.

When a token arrives at the left input, it is inserted into the left memory and its counter is calculated by trying to match it with tokens from the right memory. When a token arrives at the right input, it is inserted into the right memory and we attempt to match it with every token in the left memory, if it matches its counter is incremented by one.

Beta-test nodes Beta-test nodes are inserted for every test defined in a rule. When a token enters a beta-test node, it checks whether the token passes the test, if it does it is forwarded to the successors of the beta-test node, if it does not it is discarded.

As mentioned before, it is possible for the user to extend these tests with user-defined functions and predicates. This is very useful for providing functionality specific to the problem domain.

Terminal nodes One terminal node is generated for every rule. It executes the RHS of the rule. Tokens can only arrive at a terminal node if they have matched all patterns in the LHS of the rule.

Terminal nodes can assert new facts, by sending tokens with a + tag into the network; and retract facts by sending tokens with a − tag.

4.1.5 Speed-up of the Rete algorithm

There are several reasons why the Rete algorithm is faster than a naive algorithm that tries to match every rule to every fact in a double iteration:

- Iteration over the facts is avoided, by storing information between cycles in the memories of the two-input nodes. Instead of comparing a pattern to every working memory element one by one, we store along every pattern a list of

elements that match it. These lists are the tokens stored in the memories of the two-input nodes.

- Iteration over the rules is avoided, by compiling them into a network. This network can be seen as a tree-structured sorting network or *index*, and only when matching tokens travel through the network will the RHS be fired (when these tokens reach a terminal node).
- Nodes can be re-used. When several rules contain similar patterns, they generate identical nodes, which can be re-used.

  For example, consider the following three patterns:

```
(Point (x 2) (y 3))
(Point (x 2) (y 4))
(Point (x 1) (y 18))
```

  All of these patterns need a node to test whether the token has class "Point", so this node can be re-used for three patterns. There will also be a node that checks whether the x attribute of the point is two, this node can be re-used for the first and the second pattern.

  In most systems, there will be many patterns amongst which at least some node can be re-used. Nodes that check whether a coordinate of a point is zero for instance, or nodes for rules that contain similar patterns. In a worst case scenario, this feature delivers no speed-up, in other cases it delivers some speed-up.

  A change introduced by this feature is that it is now possible for a node to have more than one successor.

4.2 PARTE

There exist several ways to parallelize and distribute the Rete algorithm (subsection 3.2.2). This thesis extends the model of PARTE, as outlined in the Master thesis of Renaux [2012].

4.2.1 Parallelization using actors

PARTE [Renaux, 2012] is a system that implements the Rete algorithm using the actor model, similar to the system outlined by Gupta et al. [1986]. In PARTE, every Rete node is represented as an actor. The tokens sent between the nodes are passed as messages between the actors.

The Rete network is a representation of the dependencies between the tests that should be performed on facts to check whether they satisfy a rule. By converting nodes to actors, PARTE achieves an efficient parallelization of the match phase: tests

that can be done in parallel are in different, unconnected nodes; tests that depend on previous results are in connected nodes.

Because every node has at most two predecessors, there is also only a small chance of lock contention when writing a message to an inbox.

4.2.2 Focus on soft real-time guarantees

PARTE specifically focuses on providing soft real-time guarantees on the time required to detect patterns. It provides predictable upper bounds on the time required to recognize patterns.

As one of the requirements for this system is that it should be suitable for online processing (see section 2.6), PARTE is a good foundation upon which to build this system.

4.2.3 Limitations

There are some limitations to PARTE. As the system presented in this thesis builds on PARTE, these limitations also exist for it:

- No dynamic querying: it is not possible to execute a new query over the fact base at run time, as it is dispersed over the nodes in the Rete network. The fact base can only be queried by specifying rules beforehand.
- Facts need to be events: PARTE is focused on CED, and only accepts facts that have a temporal component. Some traditional use cases of expert systems are not a right fit for PARTE.
- No support for negation, nor for existential quantification: these are not supported by PARTE.

4.2.4 Sources of parallelism

There are two sources of parallelism exploited by PARTE.

First of all, *rule parallelism* means different rules can be matched simultaneously. Furthermore, through the use of Rete, different parts of the same rule can also be parallelized.

Secondly, the use of the actor model enables *pipeline parallelism*: different facts can be matched to the same rule simultaneously, in a pipeline. This is illustrated in Figure 4.3: when the first fact has processed through the first node, the second fact can enter the first node already. There is no need to wait until the first fact has arrived at a terminal node. This type of parallelism is possible because the granularity of parallelism is at the node level – which correspond to single tests – instead of at the rule level as in some other systems.

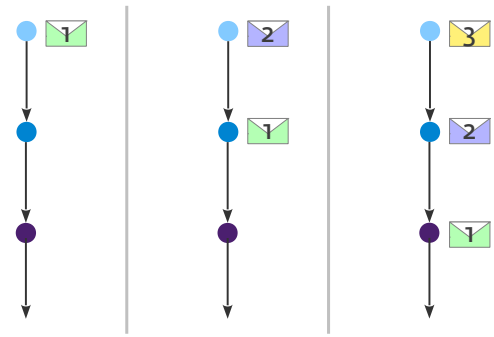


Figure 4.3: Illustration of pipeline parallelism: in the second step a second message can be processed while the first one is still being processed by the second node of the network. In the last step three messages are being processed in parallel.

Chapter 5
Distributing the Rete network

This chapter presents DPARTE, a system based on PARTE in which the Rete algorithm is spread over multiple machines with distributed memory. Its architecture is explained, and some of the issues in moving from a system with shared memory to one with distributed memory are discussed. It is also explained how to best partition the Rete network over the available machines. Finally, some limitations of this system are noted – most notably the static distribution of the nodes, which is remedied in the next chapter.

5.1 Architecture of the system

The system developed for this thesis is an implementation of the Rete algorithm. It is based on PARTE (Parallel Actor-based ReTe Engine), the parallel implementation of Rete created by Renaux [2012]. PARTE is adapted into a distributed system that can run on multiple machines, named DPARTE – "Distributed PARTE".

The system is implemented in C++ and uses the Theron library[1], an actor library that also has support for remote actors (i.e. actors can be created on different machines and communicate messages).

The architecture of DPARTE is illustrated in Figure 5.1. As in PARTE, the nodes of the Rete network are represented as actors. Multiple actors execute concurrently but maintain a single thread of control internally. In DPARTE, the actors are distributed over multiple machines. Every machine runs a single process (so there is a one-to-one relation between machines and processes) that can contain many actors. A process can contain many actors but every actor can only be in one process at a certain moment.

[1] http://www.theron-library.com

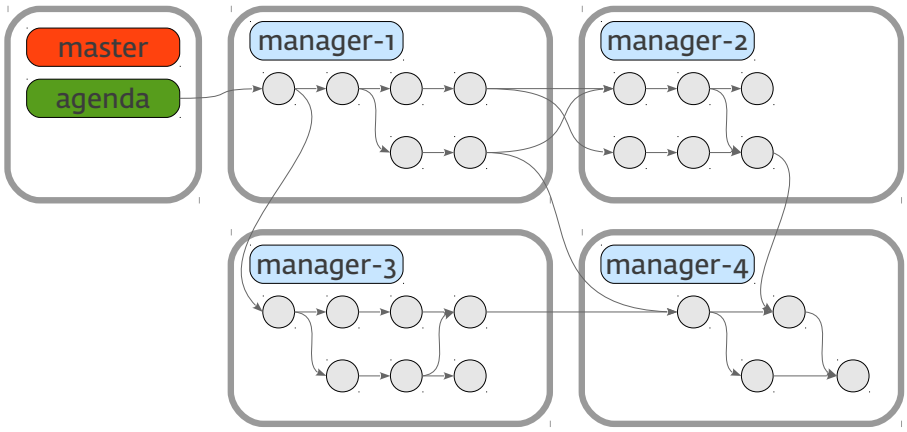


Figure 5.1: The architecture of the system. The boxes are machines: one master machine contains a master and agenda actor, and every slave machine contains a part of the Rete network and a manager actor.

Actors communicate with each other by sending messages asynchronously: every actor has an inbox associated with it, and actors wishing to communicate with it can put messages in that inbox. Messages can be sent locally and remotely, although remote messages need to be serialized (subsection 5.1.2 and section 5.2 discuss remote message sending in more detail).

5.1.1 Nodes of the Rete network

Every node of the Rete network is represented by an actor. The Rete network is a graph in which every node can have zero, one or multiple successors – in this system these can be on the same or on different machines. Every node also has one or more predecessors, which will write messages to its inbox. (In the case of a join node, there are two inboxes: a left and a right one.)

In this system, every node gets a unique name, which is determined when the program is started and the rules are compiled into the Rete network. A node knows its predecessors and successors by their unique names. A node's name is its address.

The Rete nodes are the objects that will use most resources of the system: they can save state and therefore use memory, and can do calculations which use the processor. There are different kinds of Rete nodes, as explained in subsection 4.1.4:

- **Root node**

  This node is eliminated from our system: instead of sending newly asserted facts to the root node which passes them to its successors, these facts are sent to the successors of the root node at once.

- **Constant-test node**

  A constant-test node tests features within a single fact, and passes the fact to its successors if it passes the tests. If it fails, it is deleted.

- **Two-input node** (join node)

  A two-input node – called join node in our system – tries to match two tokens. If it receives a token in its left inbox, an attempt to match the token to every token in the right memory is made. For every match, a token is sent to the successors of this node. (A similar process takes places for tokens arriving at the right inbox.)

- **Beta-test node**

  A beta-test node executes a test on variables bound to features of the facts processed by previous nodes. If the test succeeds, the token is passed to the successors of this node, else it is deleted.

  Next to a set of built-in tests such as '<', '==' or 'and' (to combine several tests), our system also supports user-defined operators. These operators could perform a number of tests, such as whether a 3D point is between two other points, whether a certain timespan overlaps with another one, whether a GPS location is within a certain distance of another one, or any other test specific to the problem domain of the application.

- **Terminal node**

  Terminal nodes execute the right-hand sides of rules. These can contain assertions of new facts, which are sent to the agenda; and calls to functions defined and implemented by the user.

Rete nodes are the units of parallelism in our system: multiple nodes can run in parallel, but inside a node there is no parallelism.

5.1.2 Communication with remote actors: RemoteSender

Every process also contains one object responsible for sending messages to, and receiving messages from, other processes, called the 'RemoteSender'. Actors in other processes are called *remote actors*; actors in the same process are *local actors*.

When a message is sent to a remote actor, the RemoteSender will take care of the serialization of the message: the message is converted from a C++ data structure – that may contain pointers to memory locations on this machine – to a uniform representation that can be read by all machines: a JSON object. Upon receival at the remote site, it is deserialized: converted back from the JSON representation to a C++ data structure.

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. In scenarios where the available bandwidth is limited, one might opt for another, more compact representation. Another consideration when choosing a serialization language is how fast a message can be serialized and deserialized: if low latency is a primary concern, a language which allows for fast serialization and deserialization will be preferable.

Because messages sent between processes might get lost, our system supports sending confirmation messages upon successful receival of a message. The sender will re-send messages as long as it did not receive a confirmation. The receiver will re-send confirmation messages if it receives a message more than once. (This is explained in more detail in subsection 5.2.2.)

5.1.3 Agenda

The system also contains a single 'Agenda' actor. This object will assert new facts into the system when they are captured outside the system (e.g. by a sensor). In DPARTE, whenever a user calls a user-defined function on the RHS of a rule, this function call and its arguments will also be passed to the agenda.

5.1.4 Manager & Master

Finally, there are two other special types of actors: a 'Manager' actor per process and one global 'Master' actor. The manager oversees his part of the network: it spawns the nodes within its process and a RemoteSender during the initialization of the program, and keeps track of them.

The master oversees the complete system. During initialization, it parses the rules that the user defined, creates a Rete network out of them, decides which nodes should move to which machine, and sends them to the managers on those machines. It also creates the Agenda, which runs in the same process as the master. While the program is running, the master will keep track of the Rete network. It knows where each node is at every moment.

In the next chapter, when nodes can move between processes, the managers and master will get more responsibilities.

5.2 From shared memory to distributed memory

In this section, a short overview is given of changes that need to be made to a parallel implementation of Rete, such as PARTE, to turn it from a system with shared memory to one with distributed memory.

5.2.1 Sending messages

When one actor sends another actor a message in a system with shared memory, the following procedure is followed by the Theron library:

1. The message is constructed by the sending actor.
2. A lock on the inbox of the receiving actor is acquired.
3. The message is inserted into that inbox (at the back).
4. The lock is released.

In the shared memory case, when a message is inserted into an inbox, this simply means a pointer to the message is added to the inbox. After this has happened, the ownership over the message has been transfered from the sending to the receiving actor.

At a later time, the receiving actor will check whether its inbox contains messages, and pop and process them one by one (in the same order as they were inserted). Checking whether the inbox contains messages and popping one needs to happen while the lock on the inbox is kept.

Using this procedure, a sending actor can only block while acquiring the lock on the inbox of the receiver; the receiving actor can only block when acquiring the lock on its own inbox. Because 1) the amount of steps that happen while a lock is kept is small (inserting or popping a pointer from a list), and 2) the amount of actors competing for a lock is small (the receiving actor and its predecessors in the Rete graph), lock contention will not a problem using this method.

In a distributed system, such a procedure does not work for various reasons:

- Messages are transfered over a network connection, so when sending a message it needs to be written to the network, and when receiving a message an object should be listening to the network.
- Messages cannot contain pointers, as these are invalid in another process. Hence, messages need to be serialized.
- The sending actor should not block while a message is being sent, nor should the receiving actor block while a message is being received.

Overall, we want to keep the overhead of sending messages over the network small for both the sending and the receiving actor. For instance, the sending actors should not compete for access to the network socket, nor should the receiving actors be blocked waiting for the receival of a message from a remote sender.

Moreover, because of the high cost of sending to remote actors, the partitioning of the Rete network should be optimized in such a way that most communication is local within one process (as explained in section 5.3). Therefore, the procedure for sending remote messages should not make sending local messages (significantly) slower.

To this end, a separate mechanism is followed depending on whether the receiver of a message is local or remote. If it is local, the message is sent as previously in a shared memory system. If it is remote, the sending actor sends the message to a separate actor that will take care of actually sending the message (this actor is the RemoteSender already described in the previous section). As soon as the sender has sent its message to the RemoteSender, it can continue its execution, e.g. processing new messages, sending more messages or doing more tests.

The RemoteSender actor will thus be responsible for sending and receiving remote messages. When it receives a message to be sent, it first serializes the message. Next, the message is encapsulated into a network message, which contains a header with information such as the message size and the IP address of the receiving machine. This network message is put in a queue, which is read by a networking library in a separate thread that sends the message.

At the other end, in a separate thread a networking library is listening to the network. When a message arrives, it puts the message in a queue, which is read by the RemoteSender in that process. The network message is unpacked, and the message in it is deserialized. The deserialized message is sent to the receiving actor, which can process the message as normally.

The complete procedure is outlined in Figure 5.2. Some notable properties of this mechanism include:

- Sending actors are not blocked while the message is serialized, nor while it is being sent over the network. Similarly, receiving actors are not blocked while the message is deserialized, nor when it is being received over the network.
- Network communication and (de)serialization happen in parallel.
- Messages sent between local actors suffer no slow-down after support for remote messages has been added.
- Whenever locks are acquired, they are only held for a short time, mostly the time needed to put an element in or remove one from a queue.

  For locks on inboxes of actors, only a small amount of other actors might be competing (generally less than ten), so lock contention is not a problem. For a lock on the inbox of the RemoteSender, any actor on the current machine might compete, which could potentially lead to some lock contention. The lock on the queue of the networking library will only be acquired by the RemoteSender, so no contention is possible there.

  In general, lock contention should not form a problem using this mechanism.

5.2.2 Coping with lost messages

Another issue present in distributed systems is how to handle slow or lost messages. Network links might fail, causing messages to get omitted, and they might

- By the sending actor:
  1. The message is constructed by the sending actor.
  2. If the receiver is local:
     (a) A lock is acquired on the inbox of the receiving actor.
     (b) The message is inserted into that inbox (at the back).
     (c) The lock is released.
  3. If the receiver is remote:
     (a) A lock is acquired on the inbox of the local RemoteSender.
     (b) The message is inserted into that inbox (at the back).
     (c) The lock is released.

- By the local RemoteSender:
  1. A lock is acquired on its inbox.
  2. A message is popped from the front of the inbox.
  3. The lock is released.
  4. The message is serialized.
  5. The message is encapsulated into a network message.
  6. A lock is acquired on the queue of the networking library.
  7. The network message is put in the queue of the networking library.
  8. The lock is released.

- By the local networking library:
  1. A lock is acquired on its queue.
  2. A message is popped from the front of the queue.
  3. The lock is released.
  4. The message is sent over the network.
  5. The message is destroyed locally.

- By the remote networking library:
  1. A message is received over the network.
  2. A function is called, which places the message in the inbox of the RemoteSender.

- By the remote RemoteSender:
  1. A lock is acquired on its inbox.
  2. A message is popped from the front of the inbox.
  3. The lock is released.
  4. The message is deserialized.
  5. A lock is acquired on the inbox of the receiving actor.
  6. The message is inserted into that inbox (at the back).
  7. The lock is released.

Figure 5.2: Procedure for sending a local or a remote message.

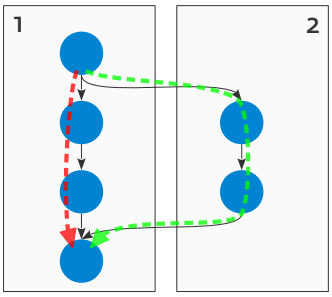


Figure 5.3: Different paths – even of same length – throughout the Rete network can have completely different latency: remote communication is much more expensive.

be restored causing new messages to arrive even though some previous ones did not. Similarly, the library used for remote allocation allocates a buffer for received messages, and in case it is full new messages are omitted. Moreover, not all paths through the Rete network are equal, a message traveling over a path between machines will take longer to arrive at its destination than a message sent over a path contained in a single process (Figure 5.3).

In the system presented in this thesis, two solutions are offered:

- Either confirmation messages are sent. Upon receival of a message, the receiver replies with a confirmation. In case the sender does not receive a confirmation after a certain period, it will attempt to resend the message.
- Or, lost messages are ignored. In a CED system with online processing, if after a certain period it is observed that a message has not been received, it might not be useful to resend the message, the time to react to it has already passed. Also in systems where messages cannot disappear, no confirmation messages have to be sent.

Confirmation messages

Using the first option, some extra logic is added to the RemoteSender for dealing with sending and receiving remote messages. The algorithm used to handle lost messages or other omissions in the network communication is also described by Tanenbaum and Van Steen [2006].

First of all, whenever a message is sent, it is tagged with a identifier that is unique throughout the system. This ID is a pair consisting of the ID of the sending process, and an increasing integer assigned by the sending process[2]. After the message is

[2]There is no problem in generating unique increasing integers in one process, as only one actor per process (the RemoteSender) will be generating them, and an actor has a single thread of control internally.

sent, it is not yet deleted from the sender, but kept in a "sent but unconfirmed" list.

Upon receival of a remote message, the receiving RemoteSender sends back a confirmation message containing the ID of the received message. It also updates a local list to indicate that the message has been received (it simply stores the ID of the message in it).

When the confirmation message arrives at the sender, it removes the message from its "sent but unconfirmed" list.

To cope with messages getting lost, the sending RemoteSender will regularly (for example every one or two seconds) iterate over its "sent but unconfirmed" list, and resend the messages in it (Figure 5.4a). This happens using an *exponential backoff*: unconfirmed messages are not resent on every iteration over the list, but only every 2^nth time. For example, if the interval for checking the "sent but unconfirmed" list is one second, attempts at resending an unconfirmed message will be made after one second, two seconds, four seconds, eight seconds, etc. until a confirmation is received (Figure 5.4b).

This mechanism of exponential backoff means that, in case a network link or a machine is saturated, and is therefore taking a long time to process messages, the resending of messages will not worsen the saturation by adding more and more messages to the queue.

Moreover, confirmation messages might get lost or only arrive at the sender after it has already resent a message. Therefore, whenever a RemoteSender receives a message, it will check whether it already got the message with that ID before, if it did the messages will be discarded, if not it is processed and its ID is added to the list of already received messages. Important to note is that, even when the message has already been received and is therefore discarded, still *another confirmation will be sent* to the sender (Figure 5.4c and Figure 5.4d).

This system guarantees that, if all messages have at least a non-zero probability of arriving, they will arrive and be processed eventually. However, there is a significant cost to this solution. First of all, the amount of communication greatly increases as confirmation messages have to be sent for every remote message. Secondly, messages have to be kept in a "sent but unconfirmed" list at the sender, which will increase memory usage. Thirdly, whenever a message is received, a check is made to see whether a message with the same ID has already been received, which is a step that can take considerable time when the process has already received many messages.

These costs are why it is possible to disable this feature, as explained in the next section.

No confirmation messages

It is also possible to ignore the problem of lost messages. Because of the high cost of ensuring the arrival of messages, the option of sending confirmation messages can be disabled.

This option might be desirable in many CED systems that rely on online processing. If sensors capture data which is sent into the system, and the system then tries to detect patterns and reacts to them, the above feature could slow down that reaction to unacceptable levels. On the other hand, it is not very useful to resend lost messages, because by the time they finally arrive, they might be outdated or it might be too late to react to them.

Furthermore, in certain systems there might be a guarantee – by a lower level component such as a networking library or protocol, or the hardware – that messages are sent reliably. In such systems as well, the confirmation messages should be disabled as they incur a high cost without any additional benefits over the guarantees of the lower level component. Similarly, in systems where a message disappearing is so rare that the benefit of checking for successful arrival of messages does not weigh up against the costs, confirmation messages should be disabled.

So, if fast reaction to new data is desired, and/or if the disappearance of messages is not an issue (because of a guarantee by a lower level component or because of the low probability), it is better to disable the sending of confirmation messages.

5.3 Partitioning the Rete network

When partitioning the Rete network over the physical machines, there are several considerations to take into account. Specifically:

- Processing power and available memory is limited per machine. Nodes with high processing power usage should be spread among the available machines. Similarly, the nodes that use most memory should be spread to avoid that a part of the program is put in virtual memory (swap) – an important principle of our system is that all data is kept in RAM so it is always quickly accessible (this is needed to satisfy the requirement of online processing [section 2.6]).
- The overhead of communication (sending messages) between different machines is *a lot* higher than communication within a single machine. Not only are messages between machines sent over a network connection, but they are also serialized and deserialized at both ends. It is therefore vital to keep tightly connected parts of the Rete network together, as well as ensuring that the most often traversed links between nodes do not cross machine boundaries.

  In many of the targeted systems, the Rete network will be composed of subgraphs that are internally tightly connected, but only loosely connected to

other subgraphs. This is also related to the idea of rules being triggered in phases, as explained in section 2.5: in one phase certain subgraphs are active while in another phase other subgraphs are active.

Typically, a Rete network becomes more 'specific' the further a token progresses in it. This is illustrated in Figure 5.5, and can also be observed in the traffic monitoring application previously shown in Figure 4.2 (page 41). Splitting the network in the right way, i.e. by putting tightly connected subgraphs on the same machine, will make a huge difference in communication overhead.

Looking at the different types of Rete nodes, we note that:

- **Constant-test nodes** contain tests that are neither CPU intensive nor require much memory. However, in many networks a number of these constant-test nodes will be chained consecutively (because at the start of a rule all the intra-element features of facts are tested, one after another). Hence, it is important to keep communication overhead low in these cases, which can be achieved by placing the chain of constant test nodes on the same machine.
- **Two-input nodes** (join nodes) use most memory of all nodes: they contain memories of previous facts, that might grow considerably. The placement of these nodes is important so as to make sure the whole program fits into physical memory – if part of the program is placed in virtual memory (swap), it will slow down considerably.

  Additionally, a two-input node can use much processing power because when a fact arrives at one of its inboxes (left or right), it is tested against all facts in the memory of the other side.
- **Beta-test nodes** have a very variable processor usage, depending on their contents (remember that user-defined operators and functions can be called). Some nodes, e.g. one doing a '<' comparison, are very simple; others can contain much more complex tests. The ones using most processor power will have to be spread most carefully over the available machines.

  Beta-test nodes are often less tightly connected to other parts of the network, when they are specific to one rule only. It might therefore be easier to isolate them from the rest of the network onto a separate machine.
- **Terminal nodes** execute the RHS of a rule. Depending on the function calls the user places there, this node might use more processor power.

In the current implementation of DPARTE, the partitioning of the Rete nodes over the available machines has to be specified manually. DPARTE parses the rules and generates the Rete network, giving each node a unique name. A table of all nodes can be printed, or a graphical representation of the network can be generated (Figure 5.6). The programmer then specifies which nodes are put on which machine as initial distribution.

5.4 Limitations

The solution presented in this chapter also suffers from some limitations. (Next to the ones it inherits from PARTE, described in subsection 4.2.3.)

No fault-tolerance Fault-tolerance is the ability of a computer system to recover from a failure. For example, if a machine crashes, a fault-tolerant system would be able to continue operating properly, possibly with a decrease in quality, but without a total break-down.

The system described in this chapter is not fault-tolerant. If a machine crashes or another critical failure happens, the system is not guaranteed to continue operating properly. This is because of the state-saving property of the Rete algorithm – state is saved in the nodes of the Rete network – and the fact that these nodes are distributed over the machines in the system. If a machine crashes, the state saved by the nodes in the memory of that machine is gone, and recovery is impossible.

We argue that this limitation poses no practical problems in the use cases envisioned for this system. We assume that the probability of a machine crashing or another critical failure happening is very low.

On the other hand, adding fault-tolerance to the system would be very costly: to assure fault-tolerance, data would need to be duplicated on multiple machines, and the cost of keeping several copies and keeping them synchronized would slow down the system considerably. Because of the requirement for online processing (section 2.6), adding fault-tolerance is not feasible.

Single point of entry All facts enter the system through the master machine. As the number of facts increases, the bandwidth of the master could become saturated. The throughput will therefore be limited by the amount of facts the master can accept per time unit. An improved system could contain multiple entry points that assert facts into the network, or alternatively facts could – depending on their type – be sent to the correct entry nodes directly.

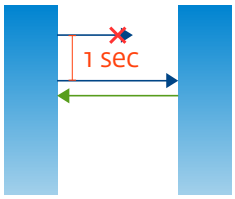
Static distribution The distribution of the Rete network over the machines, as laid out in this chapter, is static. This means it is determined at the start of the program, and does not change during its execution. There are several problems with this.

First of all, during the execution of the program the processor and memory usage of the different nodes in the Rete network might change. This means that, while at one point in time a certain partitioning of the Rete nodes over the machines will be optimal, at another moment another partitioning is better. In the architecture described here, one partitioning has to be chosen at the start, but this one will not be optimal during the complete run time of the program.

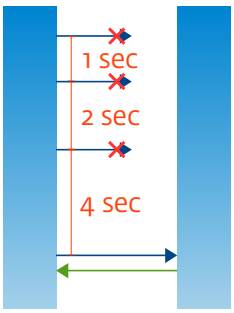
Secondly, the fact that the partitioning is made at the start of the program means it is impossible to add extra machines to the system at run time. In a practical application, if we notice that while the program is running it taxes the machines up to their limits, it would be useful if the system could be extended with additional machines at run time.

This limitation is one than can be remedied. In the next chapter, the nodes of the Rete network will become *mobile actors*. These are actors that can move at run time, they constitute a form of code mobility. By moving the actors at run time, the system contains *dynamic* load balancing, and and it becomes possible for machines to be added and removed[3] at run time.

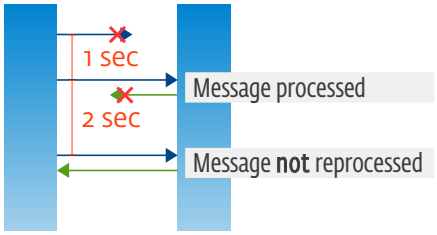
[3] Note that removing machines is only possible in a controlled fashion: when a machine is removed all its actors first have to be moved to other machines. In other words, the ability to remove machines at run time is not a form of fault-tolerance.



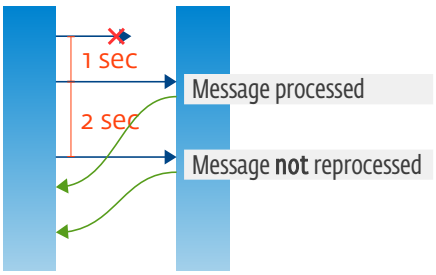
(a) Messages get resent until a confirmation message is received.



(b) Messages are resent with an exponential backoff.



(c) A confirmation message is lost, and as a result the message arrives twice at the receiver. The second message is discarded, but another confirmation is sent.



(d) A confirmation messages travels slowly, and only arrives at the sender after it has already resent the message. The second message is discarded, but another confirmation is sent (because we do not know whether the first confirmation successfully arrived).

Figure 5.4: Different scenarios demonstrating how messages can be resent. The blue arrows represent the message (always the same message with the same ID), the green arrows are confirmation messages.

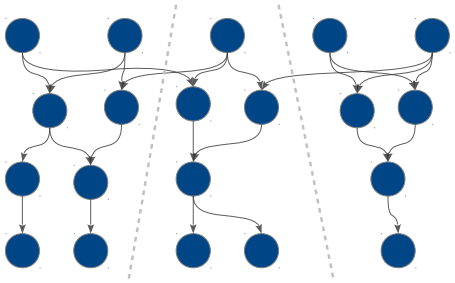


Figure 5.5: Example of how a Rete network could be split over three machines: the striped lines indicate how to split the network. As you can see, the farther 'down' into the network, the more specific rules become, which means there will be less connections to other parts of the network. As a result, there often exist ways to partition the network which minimize inter-machine communication.

Name	Type	Proc	Successors
AdapterNode_14	2	0	JoinNode_16
AdapterNode_21	2	0	JoinNode_22
AdapterNode_26	2	0	JoinNode_28
AdapterNode_34	2	0	JoinNode_36
AdapterNode_4	2	0	JoinNode_6
AdapterNode_46	2	0	JoinNode_48
AdapterNode_55	2	0	JoinNode_56
BetaTestNode_10	5	0	TerminalNode_11
BetaTestNode_29	5	0	JoinNode_30
[...]			
ConstantTestNode_52	1	0	JoinNode_58
ConstantTestNode_53	1	0	JoinNode_56
ConstantTestNode_54	1	0	AdapterNode_55
JoinNode_16	3	0	JoinNode_17
JoinNode_17	3	0	TerminalNode_18
JoinNode_22	3	0	TerminalNode_23
JoinNode_28	3	0	BetaTestNode_29
JoinNode_30	3	0	TerminalNode_31
JoinNode_36	3	0	JoinNode_38
JoinNode_38	3	0	BetaTestNode_39
JoinNode_42	3	0	TerminalNode_43
JoinNode_48	3	0	BetaTestNode_49
JoinNode_50	3	0	TerminalNode_51
JoinNode_56	3	0	BetaTestNode_57
JoinNode_58	3	0	TerminalNode_59
JoinNode_6	3	0	BetaTestNode_7
JoinNode_8	3	0	JoinNode_9
JoinNode_9	3	0	BetaTestNode_10
TerminalNode_11	6	0	
TerminalNode_18	6	0	
TerminalNode_23	6	0	
TerminalNode_31	6	0	
TerminalNode_43	6	0	
TerminalNode_51	6	0	
TerminalNode_59	6	0	

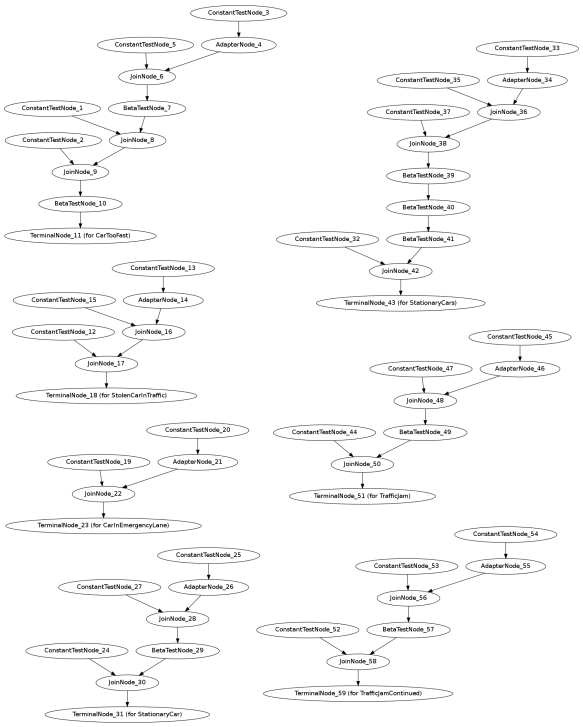


Figure 5.6: Table and graph of nodes in the Rete network generated by DPARTE, for the example of section 2.3. The graph is generated automatically using Graphviz (http://www.graphviz.org).

Chapter 6
Dynamic load balancing through mobile actors

In the previous chapter, Rete nodes were represented as actors and distributed over multiple machines, to cope with the increasing processing power and memory demands of large-scale problems. At the end of that chapter, it was suggested that statically distributing the actors over the network at the start of the program is not optimal, therefore this chapter introduces dynamic load balancing.

To implement dynamic load balancing, the actors become "mobile". They will be able to move between machines at run time, so that the processing power requirements and memory usage of the algorithm can be optimally distributed over the available hardware.

This chapter starts by introducing mobile actors, explaining how they are implemented in this system, and discussing their properties. Next, they are applied to solve the load balancing problem, and the load balancing strategy used in this system is presented.

6.1 Mobile actors

6.1.1 Making Rete nodes mobile actors

As defined in section 3.3, mobile actors are actors that can move between computational environments. In our system, the nodes of the Rete network will become mobile actors, and they will move between machines. Whenever it is decided that a node should move (how this is decided will be discussed later, in section 6.2), the node's execution will be paused, the node will move to another machine, and execution is resumed there.

As also explained in section 3.3, when a mobile actor moves three elements move:

its code, its internal state and its execution state. Concretely, in our system, every instance of the program on all machines, will contain the code necessary to create a node of all node types (constant-test, two-input, beta-test and terminal nodes). When an actual node is moved, this is how the three elements it is composed of move:

1. Its **code**

   The code component of a node is one of the following:

   - A (list of) test(s): for a constant-test node, a list of intra-element features to test; for a two-input node a list of inter-element features; for a beta-test node a user-defined test (possibly containing variables and user-defined operators and functions).
   - A list of instructions: terminal nodes contain a list of instructions, such as assertions of facts and calls to user-defined functions.

   This code is written by the user and embedded in his rules. The compiler, when it generated the Rete network, moved this code into the appropriate nodes.

2. Its **internal state**

   Constant-test, beta-test and terminal nodes contain no internal state. Two-input nodes are the only type of node that contain internal state, namely their left and right memories, which contain tokens that have been sent to their left and right inboxes. These need to be moved.

3. Its **execution state**

   Actors continually take a message from their inbox, process it, take another message, process it, and so on. We can think of these steps as "turns". Consider these turns *atomic*: they are indivisible, they cannot be broken into parts. In other words, if an actor moves, it will always move between the processing of two messages, never *during* the processing of a message.

   In this case, the execution state of an actor – at what point during its computation it was paused – is fully represented by its inbox. The inbox of the actor represents the current state of the execution: it represents the computations that still have to be done in the form of the messages that still have to be processed. Hence, to move the execution state of an actor, it suffices to move its inbox.

   However, this is not the exact implementation used in this thesis. As explained in the next section, if no special precautions are taken it is possible that the predecessors of a node will be sending it messages *while* it is moving. This is of course not desirable: messages sent to a moving node will not arrive because the node does not exist at that moment. Therefore, the predecessors of a node will be informed before it moves. As a result, it will (as also ex-

plained later) automatically be the case that an actor's inbox is empty when it starts moving, so actually moving the inbox is not needed anymore.

It is then necessary though, immediately after the actor resumes at its new location, that it informs its predecessors that the movement is complete. A notification is sent to the actor when it resumes so it can do this, you could consider this notification a way of telling the actor what its execution state is (which is "just resumed").

6.1.2 Buffering of messages during a move

When an actor moves, it will disappear on one machine to later re-appear on another machine. This means that there will be a period of time during which the actor *does not exist*. Messages sent to the actor during that period will not arrive, which is why measures have to be taken to assure that the predecessors of the actor will not be sending it messages while it is moving.

Duplicating the moving actor (i.e. first create a copy at the new location and then destroy the original actor, thus allowing it to exist in two places simultaneously) is also not possible, as actors contain state which is modified by the incoming messages, and in a scheme with duplicated actors each of them will only receive a subset of the total amount of messages sent to the node. For example, when a join node receives a token at its right inbox, it will save this token in its right memory, and when later a token arrives at its left inbox, this token is matched against those in the right memory. However, in case this node is duplicated, these two tokens can arrive at different copies of the actor, and they will not be matched.

For those reasons, an actor moves by destroying it at its original location and recreating it in its destination, and during the time it does not exist messages sent to it have to be buffered. The predecessors of the actor will need to be informed before moving to node so they can start buffering, and after the move has completed they will be asked to send their buffered messages to the new location.

Figure 6.1 contains an overview of the complete procedure followed when moving a node. In the figure, machine 2 contains a node that will move to machine 3. Machine 1 contains a predecessor of the moving node.

1. A move is always initiated by the master, which contains a load balancing algorithm (subsection 6.2.2) that decides when to move a node. It notifies the node that it should move, and its destination.
2. When the node receives this message, it informs all of its predecessors that it will move.
   (a) When the predecessors receive this message, they switch an internal flag to indicate that messages destined for that actor should be buffered locally instead of being sent over the network to the moving actor.

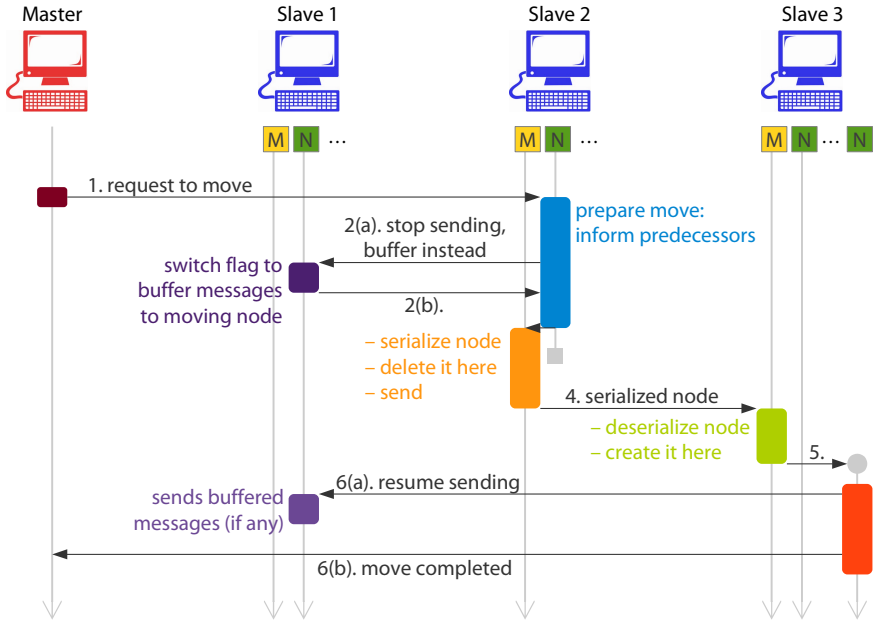


Figure 6.1: Procedure to move a node to a different machine.
Every slave machine contains one local manager (M) and several nodes (N). A node on slave 2 moves to slave 3. The node on slave 1 is a predecessor of the moving node: when it receives message 2(a) it starts buffering messages destined for the moving actor, when it receives message 6(a) it sends the buffered messages to the moved node.

(b) The predecessors reply with a confirmation that they will now buffer.

3. Only when the moving actor has received a confirmation from *all* of its predecessors that they are now buffering, it can start the actual move. It sends a message to the local "manager" to start the move.
4. The local manager (of the origin) serializes the node, destroys it locally, and sends the serialization to the manager of the destination machine.
5. The manager of the destination deserializes the node and creates it there. It notifies the node that it has just been resumed.
6. The node, at its new location, notifies it predecessors that it has resumed and they can send their buffered messages. It also notifies the master that the move is complete.

6.1.3 Limitations

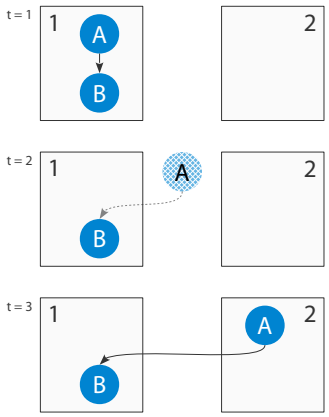


Figure 6.2: Actor A moves from machine 1 to machine 2. At time t = 2, while it is being sent between the machines, it does not exist on any machine. If at t = 2 actor B would want to start moving, this is not possible: it can not inform A that it will move.

A consequence of the procedure presented here is that no two actors can move simultaneously. If we would allow two actors to move simultaneously, and one of them was a predecessor to the other, then the message sent by the successor asking its predecessor to buffer messages would not arrive at that actor, because it does not exist as it is moving too. This is illustrated in Figure 6.2: if at time t = 2, actor B would initiate a move, it would need to inform its predecessors about this. However, sending a message to actor A is not possible as it does not exist at that moment.

Because all moving operations are initiated by a centralized master node, preventing two actors from moving simultaneously is easy: the master node simply waits

until one movement has completed before starting another one. This is why, in the previous section, the last action of a resumed actor is to inform the master of the completion of the movement (message 6(b) in Figure 6.1).

While this may seem as a limitation at first, prohibiting simultaneous movements is actually not a problem. Because actors will move as a way of remedying load imbalances, and moving actors has a temporary effect on how the load is spread, it is not a good idea to move two actors at the same time anyway. Even more, after an actor has moved it is probably a good idea to wait some time before moving another actor: messages for the moving actor were buffered, which caused the memory usage of its predecessors to rise during the move, and the actor that just moved will be processing more messages than usual right after the move. After a move, it is therefore better to wait until the system reaches an equilibrium state again before more load balancing actions are taken.

6.1.4 Code mobility properties of the system

As explained in section 3.3, Fuggetta et al. [1998] make a distinction between strong and weak mobility: strong mobility is the ability of a system to allow migration of both the code and the execution state, while weak mobility only allows the migration of code and some initialization data, but not the execution state. The system presented here supports strong mobility according to this definition: an actor is suspended, moves to another machine, and resumes execution there – it restarts from the exact point at which it was suspended, and therefore its execution state moves as well as its code.

As also described in section 3.3, there are two mechanisms for code mobility: migration and remote cloning. Our system uses migration: an actor is suspended, transmitted to another machine, and resumed there. This migration is reactive: a centralized node determines when and where migrations should happen (as opposed to proactive, where the time and destination of the move are decided autonomously by the moving actor).

	Data context	Control context	Resources context
Weak mobility	✗	✗	✗
Semi-strong mobility	✓	✗	✗
Strong mobility	✓	✓	✗
Full mobility	✓	✓	✓
This system	✓	✓	✓

Figure 6.3: Types of code mobility according to De Meuter [2004], and how this applies to the system in this thesis.

De Meuter [2004] makes a distinction between four types of code mobility: weak,

semi-strong, strong and full mobility. This distinction is based on which parts of the computational context move:

- The *data context* represents the variable bindings accessible by and allocated to the moving program. This is its internal state as defined in subsection 6.1.1, which moves along with the actor in our system.
- The *control context* consists of the status of the computation, i.e. at what point during the computation the program was paused. As explained in subsection 6.1.1, in a system with mobile actors this is represented by the inbox of the moving actor.
- The *resources context* consists of the bindings between the moving program and memory that is not owned by the program alone. For example: bindings to operating system resources, opened files, a connection to a database, etc.

  In our system, an actor only has one kind of binding: it is connected to other actors, to which it can send messages and from which it can receive messages. These bindings happen through the addresses of the actors. Every actor has a list of addresses of its successors and predecessors in the Rete network. When an actor moves, this list moves along with it, so in this way its resources context moves as well.

Based on these characteristics of the system, and the table in Figure 6.3, we conclude the system implemented here supports full code mobility.

6.2 Load balancing

6.2.1 Dynamic load balancing

A key issue in a distributed system is how to partition the problem over the available hardware, i.e. how to divide the work over the machines. The goal of **load balancing** is for each machine to perform an equitable share of the work load [Cybenko, 1989], which ideally makes the efficiency of the computation optimal and the run time minimal [Watts and Taylor, 1998].

In some cases, such as an operation on a dense matrix, it is possible to predict the work load a priori. Hence, the partitioning of the work can be built into the program. This a priori partitioning of the work is called **static** load balancing.

In other cases, it is not possible to estimate the work load distribution a priori: only when the program is executed does it become clear how the work is partitioned over the machines. This is also the case in our application, and in this case **dynamic** load balancing is required.

Load balancing consists of two subproblems [Watts and Taylor, 1998]:

1. The **problem decomposition** asks how to partition the problem into tasks.

2. The **task mapping** determines how these tasks are mapped onto machines.

As already discussed, the problem decomposition in our system consists of making every node in the Rete network a separate actor and moving tokens between them as separate tasks. The problem of task mapping is further discussed in this section.

6.2.2 Load balancing strategy

Four-phase load balancing model

Willebeek-LeMair and Reeves [1993] and Watts and Taylor [1998] describe a respectively four-phase and five-phase model for load balancing. These are the phases of the four-phase model[1]:

1. **Load evaluation**

   The load of every machine in the system is estimated. These values are used to determine whether there are any *load imbalances*.

2. **Profitability determination**

   An *imbalance factor* is calculated to estimate the 'degree' of load imbalance for every machine. If this value exceeds a certain threshold, work should be migrated.

   The imbalance factor weighs the amount of speed-up achievable by eliminating the imbalance against the overhead of doing load balancing. It should determine whether load balancing is 'profitable' or not at this moment.

3. **Task migration strategy**

   The source(s) and destination(s) of the task migration are determined, in other words, it is determined which machines are overloaded and which are underloaded. It is also determined how much work should ideally be transfered between them (without selecting which exact tasks to move yet).

4. **Task selection strategy**

   The most suitable task(s) from the source machine(s) are selected for migration.

Load evaluation

The first phase of the load balancing model is concerned with evaluating the load of every machine. In our system, three factors should be taken into account:

- Processor usage: reported by the operating system.

[1] The five-phase model simply adds a final phase in which the actual migration happens.

- Memory usage: either as reported by the operating system for the total machine, or by counting the saved state by the individual nodes.
- Network usage: this can be measured by the amount of remote messages sent (and their size).

One technical issue is measuring the contributions of the different actors in the same process to these: for instance, the operating system can tell us the CPU usage of the process, but not directly how much was used by each separate actor.

There are also some metrics that indirectly measure these factors, such as the length of the inboxes, which will grow if there is not enough processor power available to process messages at the same rate at which they arrive. In the current implementation, this method is applied. The "Manager" of each process measures the length of the inboxes of the actors on its machine, and it sends these results to the master.

Profitability determination

To determine whether there is a load imbalance, we check whether the observed load exceeds certain thresholds. If a machine has a processor, memory or network usage above a certain limit, it is considered overloaded; similarly if the load is below a certain limit the machine is underloaded.

In the current implementation an actor is considered overloaded if its inbox contains more than 50 messages. A machine is considered overloaded if it contains five or more overloaded nodes. This is because the experiments run on machines that contain four cores, so if more nodes are overloaded than there are cores the machine will not contain sufficient processing power to run them all simultaneously. A machine is considered underloaded if it contains two or less overloaded nodes. Of course, in different situations other parameters might be more applicable.

Predicting the profitability of load balancing is a more difficult problem, for this estimates of the speed-up achieved by load balancing and the cost associated with it have to be made. One way to do this is based on previous examples: by keeping a history of past load balancing actions, an attempt could be made to predict the speed-up and cost associated with new actions.

In the current algorithm, because the threshold at which a node is considered overloaded (50 messages in its inbox) and only one move is allowed at the same time, no separate profitability estimation is done: as soon as an overloaded machine is found, load balancing will take place.

Task migration and selection strategy

Finally, the source and destination machine have to be determined, and which node(s) to move between them. The source machine in the current implementation is simply the most overloaded machine.

To determine which node to move, there are several considerations to take into account. First of all, the contribution of every node to the total load on the machine should be taken into account: it is more beneficial to move a node that causes a large load. However, the algorithm should also take into account how tightly connected this node is to other nodes on the same machine: links between the moved node and other nodes on the same machine will turn from local communication to much more expensive remote communication. It is therefore more beneficial to move nodes that are only loosely connected to other nodes on the machine.

In the current implementation, the overloaded node with the *least* amount of messages is chosen to move. So, this will be a node that both contains more than 50 messages in its queue, but at the same time is not the most overloaded node on the machine. This is because, while moving a node, messages sent to it need to be buffered, and as a result when the nodes re-appears on the destination machine it will be even more overloaded than it was before – at least for a short time. Instead of taking the most overloaded node and causing it to be overloaded even more, the algorithm chooses the least overloaded amongst the overloaded.

The other overloaded nodes on the overloaded machine of course also benefit from the node moving, and this as soon as it has disappeared from that machine. So, using the current algorithm, as soon as a machine contains five overloaded nodes, load balancing will happen and the least overloaded amongst the overloaded nodes on the overloaded machine will move – so the other four (more) overloaded nodes that remain on the machine again can use the four cores of the machine all the time.

The destination chosen by the algorithm will be a randomly selected underloaded machine. A smarter algorithm might chose a machine that has connections to the node being moved, so that some communication that was previously remote now becomes local (more details on future work are discussed in subsection 8.2.1).

6.2.3 Classification of load balancing model

Zaki et al. [1995] describes a classification of load balancing strategies along two axes:

1. A load balancing strategy is either *global* or *local*: in global strategies, decisions are made using information about the complete system, while in local strategies work moves based on information about a part of the system. Our strategy is global: information about every machine is known and used to make load balancing decisions.

2. The load balancing can be *centralized* or *distributed*: in centralized systems there is one load balancer on a master machine, in distributed systems load balancing decisions can be made by every machine separately. Our strategy uses a centralized load balancer on a master machine.

6.3 Summary

In this chapter, mobile actors were introduced, and applied to DPARTE to add dynamic load balancing.

The dynamic load balancing model used by DPARTE is global and centralized: one master machine gathers information about all other machines, and determines whether load balancing is necessary and if so which nodes should be moved and their destination.

Mobile actors are the technique used to achieve the load balancing: an overloaded node of the Rete network can move from an overloaded to an underloaded machine. During this process, all elements of the actor, i.e. its code, its internal state and its execution state, are transfered. This means the system supports full code mobility.

The movement of actors is completely 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.

Chapter 7
Evaluation

In this chapter, the system presented in the previous two chapters is evaluated. First, the experimental methodology and setup is described, next a series of experiments validates whether the system meets the requirements described in section 2.6.

7.1 Methodology & setup

7.1.1 Hardware and software setup

All experiments ran on a network of commodity computers, which each contain a quad-core processor and 8 GB memory. The computers are connected in a Local Area Network using wired connections at 1000 Mbit/s. All code was compiled using gcc 4.6.3 with optimization flag -O3. The specifications of the hardware and software are given in the table in Figure 7.1.

7.1.2 Experimental setup

Each experiment compares a number of configurations, which consist of a list of benchmarks (described in subsection 7.1.5) and a list of parameters (listed in subsection 7.1.4), by measuring a variable (one of the list in subsection 7.1.3). ReBench[1] executes the benchmarks based on the defined configurations.

Based on the methodology proposed by Georges et al. [2007], 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.

7.1.3 List of measured variables

In the experiments, two variables are measured:

[1] https://github.com/smarr/ReBench

Hardware	
Processor	
Type	Intel Core i5-3570
Architecture	64-bit
Clock frequency	3.40 GHz
Number of cores	4
Cache	6 MB
RAM	
Size	8 GB
Network (ethernet)	
Speed	1000 Mbit/s
Software	
Operating system	
OS version	Ubuntu 12.04.2 (Precise Pangolin)
Kernel version	Linux 3.2.0-43-generic
Compiler	
Compiler version	gcc 4.6.3
Optimization flags	-O3

Figure 7.1: Specifications of the computers on which the experiments were performed.

- The **run time** of a benchmark: in this case, the time between asserting the first fact in the benchmark and successfully having processed all of them is measured. In other words, the experiments measures the time it takes for all of the facts to process through the Rete network.
- **Throughput**: here, the number of facts that can be processed by the system per time unit is measured. The throughput is measured indirectly, by dividing the total run time by the amount of facts asserted by the benchmark.

The results are displayed using beanplots [Kampstra, 2008] or violinplots [Hintze and Nelson, 1998]. These allow a quick visual comparison of the results, their distribution, and their significance. The median value is indicated with a red dot, the arithmetic mean with a blue line.

7.1.4 List of parameters

These are the parameters that are varied in different experiments:

- **Number of machines used**: when measuring the scalability of the system, the number of machines is increased to measure the effect on the throughput. In an ideal case, doubling the amount of machines doubles the throughput.
- **Confirmation messages enabled/disabled**: since the system supports sending confirmation messages (and resending messages in case no confirmation is received), the system is evaluated with this option enabled or not. It is expected that enabling confirmation messages causes a decrease in performance, because (as explained in subsection 5.2.2) (1) for every received message a confirmation is sent and received, (2) for every received message it is checked whether the same message has already been received before, (3) for every sent message it is regularly checked whether a confirmation has been received already, if not it is resent.
- **Dynamic load balancing enabled/disabled**: the system introduced in Chapter 6 with dynamic load balancing is compared with the one without load balancing from Chapter 5.

7.1.5 Explanation of different benchmarks

Below is a description of the benchmarks used to evaluate this system. The set of benchmarks consists of, on the one hand a set of benchmarks created to evaluate PARTE [Renaux, 2012; Renaux et al., 2012], and on the other hand some benchmarks to evaluate the specific properties of the system developed for this thesis. The descriptions of the former are based on those in Marr et al. [2013], the latter are indicated with ($\star$).

- Simple tests (100, 500)

  The Rete network in these benchmarks consists of a linear chain of (100 or 500) test nodes, each of which does a very simple test. 5000 facts are asserted, travel through this chain, and the last one will trigger the end of the benchmark.

  This test will measure the communication overhead, and the benefit of pipeline parallelism (subsection 4.2.4). A parallel implementation of Rete is expected to improve the execution time, as it will benefit from pipeline parallelism, but is on the other hand expected to have an overhead related to communication that does not exist in a sequential implementation. A distributed version will have a larger communication overhead.

- Complex tests

  This benchmark is similar to the Simple Tests, except here each test is more complex: it contains a boolean expression that consists of nine nested binary expressions that add numbers.

  This test will cause less communication overhead, and measures the speed of the expression evaluation. It will also benefit from pipeline parallelism. Because the system presented here builds upon PARTE and re-uses its expression evaluator, no speed-up compared to PARTE is expected.

- Heavy tests (16, 32, 64, 128)

  The Rete network in this benchmark consists of a linear chain of N tests (with N = 16, 32, 64 or 128), each of which is computationally intensive. The tests represent the computational cost of using machine learning algorithms such as a Hidden Markov Model [Wilson and Bobick, 1999] inside a rule. Each test is independent of the others.

  In this tests, parallel and distributed systems are expected to scale perfectly, as the amount of communication is minimal and the independent computations can be distributed over the available hardware.

- Simple tests in parallel (10 times 100 tests)

  In this benchmark, the Rete network consists of 10 independent chains of 100 test nodes, i.e. the network of the "Simple Tests" benchmark is repeated 10 times in parallel. In a sequential implementation, this benchmark should perform similar to the "Simple Tests" benchmark; in a parallel implementation however the amount of communication will increase further, leading to a larger overhead.

- X times Y heavy tests in parallel (10 times 8 tests, 15 times 16 tests)

  This benchmark consists of X independent chains of Y heavy tests, i.e. it consists of X independent instances of the "Heavy Tests" benchmark with Y test

nodes. It simulates use cases where several machine learning algorithms execute in parallel. It is again a case where a parallel or distributed system should scale ideally. (The 15 times 16 heavy tests in parallel benchmark is shown and described in more detail in Figure 7.4.)

- Joining tree

  This benchmark consists of rules that match the facts asserted in the benchmark, which assert facts of other types that are matched by other rules that assert yet other fact types. It simulates a scenario in which low-level rules match facts that produce more high-level facts, which in turn match other rules that produce even more high-level facts; i.e. a complex event hierarchy is created.

  This benchmark on the one hand can benefit from parallelism as multiple rules can be evaluated simultaneously, but on the other hand there is only few computation and more communication that causes overheads.

- Search

  In these benchmark, the Rete network contains 50 rules that perform computationally expensive tests but fail to match, and one rule that immediately matches the last fact asserted in the benchmark, without any computation. For a sequential implementation of Rete, this is a worst-case scenario: all computationally expensive rules execute and fail, and the last 'easy' rule succeeds. A parallel implementation can terminate as soon as it has matched this last rule, and therefore execute much quicker.

- 20 rules in 10 phases ($\star$)

  This benchmark consists of a Rete network of 20 independent rules, each of which contains a computationally expensive node. Facts are asserted that travel through only one of the nodes, in several phases: in a first phase facts travel through nodes 1 to 3, in the next phase through nodes 1 to 5, then 3 to 7, and so on until the last phase in which they travel through nodes 19 and 20 (see Figure 7.7a). Every phase consists of 10 000 facts.

  The benchmark represents use cases in which facts are asserted in phases, e.g. traffic cameras that watch traffic jams (first cameras 1, 2 and 3 will see the jam, later cameras 2, 3 and 4 see the jam, and so on) or surveillance cameras watching moving crowds (as the crowd move different cameras are more active, Figure 7.7b).

  This benchmark was specifically created for this system, its goal is to test the dynamic load balancing. It is expected that, during the first phases, the nodes most active at that point will be distributed over the available machines evenly, while in a later phase the nodes most active then might move.

7.2 Comparison to PARTE

7.2.1 Setup

The first experiment is a comparison of DPARTE to PARTE, using the benchmarks created for PARTE. These tests ran on one machine, for PARTE this means one process on one machine, for DPARTE a master process and a single slave process were started on the same machine.

First of all, it should be expected that DPARTE performs worse than PARTE in all of these benchmarks: the only difference between the two systems is the approach used to create actors and send messages between them. PARTE uses lock-free inboxes and a custom scheduler, and sends messages as pointers whenever possible[2]. DPARTE uses the Theron library, in which sending a message involves acquiring a lock on the inbox of the receiving actor, and copying the message.

The advantage of DPARTE in contrast to PARTE lies in the distribution over multiple machines, and as this experiment only ran on one machine it does not exploit the benefit of DPARTE. It is useful as a baseline for the next tests: this experiment measures the overhead caused by switching PARTE from its custom scheduler to the Theron library, and the results achieved here can be used as a reference for the experiments in the next sections.

As DPARTE only serializes messages sent between processes, in this experiment only the messages sent between the master and the one slave are serialized, but not those sent within the slave. In other words, the facts initially asserted in the benchmark are serialized and deserialized, as well as the facts asserted by the RHS of a rule, but no other messages.

7.2.2 Results

The results are shown in Figure 7.2 and Figure 7.3. The first thing to note is the bad performance for the "simple tests" benchmarks: DPARTE is about 20 times (for 101 simple tests), 30 times (501 simple tests) and even 60 times (10 times 101 simple tests) slower than PARTE. This is logical taking into account the nature of these benchmarks: the actual work done in every node is very small (as shown by the fact that the run time for 101 simple tests for PARTE is only 20 ms), and most time is spent on communication. As the ratio of communication compared to computation increases, the performance of DPARTE decreases, it can be concluded that the communication overhead of DPARTE is much larger than that of PARTE.

[2] For a node with n successors, PARTE makes n-1 copies of the message and sends one successor a pointer to the original and the others pointers to clones. So, if a node has only one successor, no copy needs to be made.

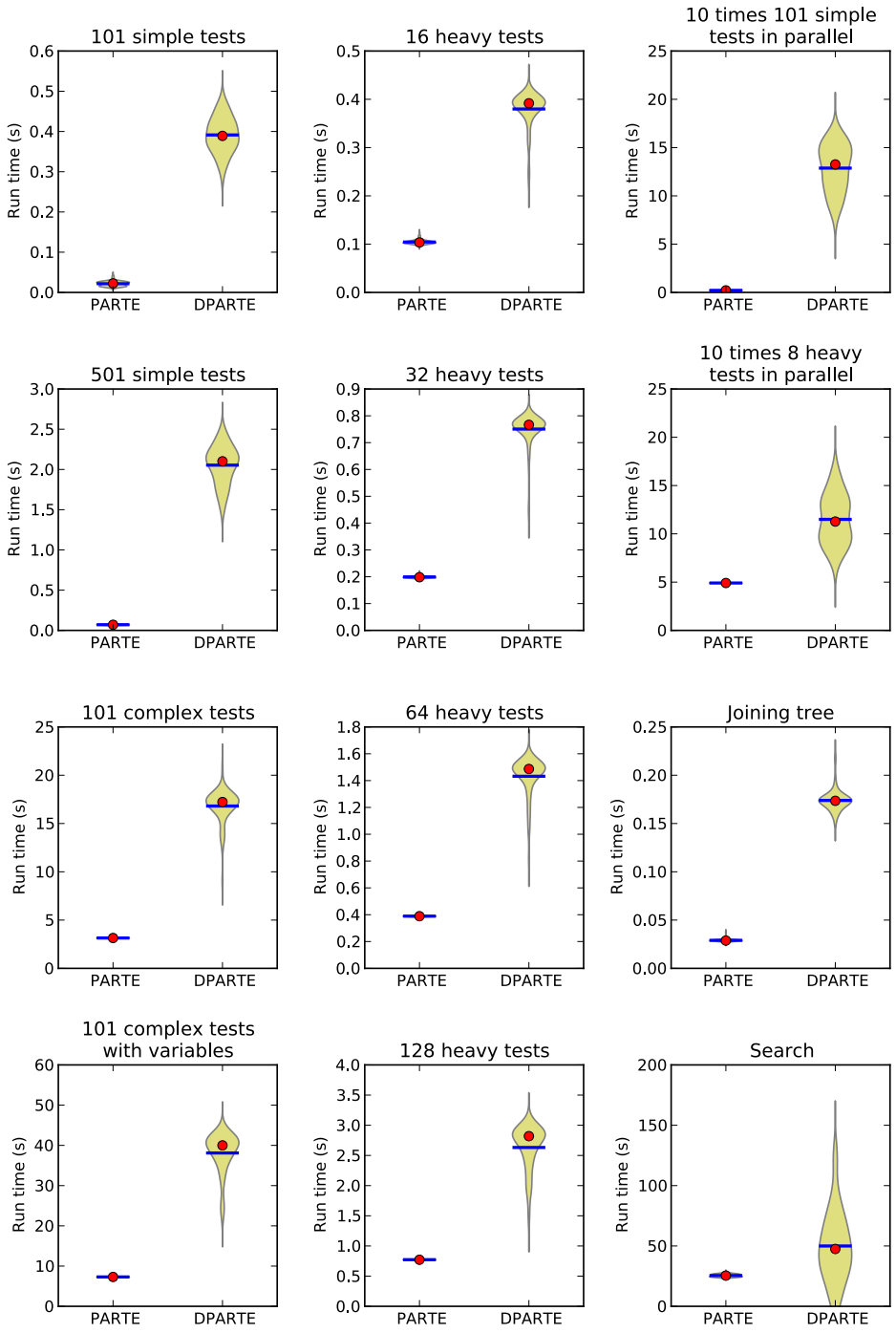


Figure 7.2: Comparison of DPARTE to PARTE. A table with results can be found in Figure 7.3.

Benchmark	Run time (ms)		Slow-down
	PARTE	DPARTE	
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
101 complex tests with var.	7,302.1± 3.0	38,111.1±1,117.2	5.22
10×101 simple tests in par.	212.8± 0.6	12,875.7± 550.8	60.49
10×8 heavy tests in par.	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

Figure 7.3: Comparison of DPARTE and PARTE.

This line of thinking also extends to the other results: the complex tests and joining tree benchmarks are slightly more computationally expensive, but still contain quite a lot of communication; the heavy tests contain most computation and least communication. The most computationally demanding benchmark, 10 times 8 heavy tests in parallel, is 2.3 times slower in DPARTE as in PARTE.

There are some other overheads still present in that case, mainly related to the setup of DPARTE. As explained in chapter 6, the architecture of DPARTE consists of a master machine and several slave machines. In this experiment, there was only one slave, and the master and slave both ran in separate processes on one machine. However, communication between the master and slave processes is still over a network socket (though within the same machine), and messages sent between the two are still serialized.

Furthermore, the experimental setup is slightly different for both systems. In PARTE, the time is registered at the start of the program, then all facts in the benchmark are asserted, and after the last fact has traveled through the Rete network it triggers the end of the benchmark. In DPARTE, the time is registered at the start of the program by the master, it asserts all facts in the benchmark, and they are sent to the slave process. The slave contains the Rete network that processes the facts, and after the final fact has triggered the end of the benchmark, it sends a message to the master, which only registers the time at the moment it receives that message. In other words, DPARTE has additional overheads in its experimental setup because it uses multiple machines.

Lastly, a note should be made about the different schedulers in the two systems. As can be seen, the variance of the results for PARTE is much smaller than that of

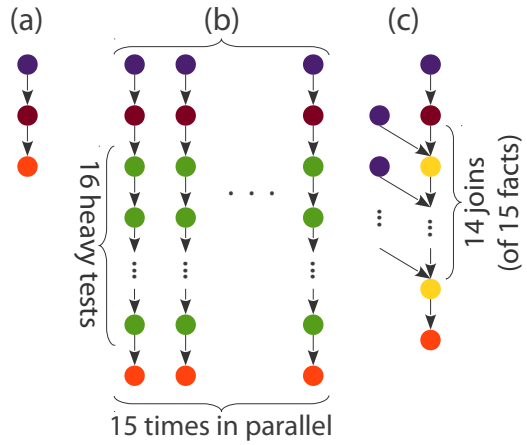


Figure 7.4: Rete network for "15 times 16 heavy tests in parallel" benchmark. Rule (a) takes the facts asserted in the benchmarks (of type F) and asserts 10 new facts (of types F0 to F14), which each move into one of the nodes of part (b). (b) contains 15 independent rules that process facts of types F0 to F14. Each of these rules does 16 computationally heavy tests, after which it looks at the timestamp of the fact and if it is the last fact of the benchmark, the terminal node asserts a fact of type JOIN with attribute 0 to 14 respectively. Rule (c) joins these 15 facts of type JOIN and triggers the end of the benchmark.

DPARTE. This is due to how work is scheduled in the two systems. In PARTE, there is one central non-blocking queue that contains all work, whenever a message is sent it is put in this queue, and worker threads take work from this queue whenever they have finished their previous task. In DPARTE, there is one inbox per actor, when a message is sent it is put in the inbox of the receiving actor and that inbox is put in a queue of inboxes to process; a pool of worker threads processes that queue.

7.3 Scalability of distributed system

7.3.1 Setup

In this experiment, the scalability of DPARTE is assessed, by measuring the throughput as the amount of machines is increased. Two variants of DPARTE are compared, one using confirmation messages and one without.

It uses the "15 times 16 heavy tests in parallel" benchmark, the Rete network corresponding to this benchmark is shown and explained in Figure 7.4. For this experiment, the rules are manually distributed over the machines, in the following way. In case of 16 machines, every rule of part (b) of Figure 7.4 is on a separate machine, the

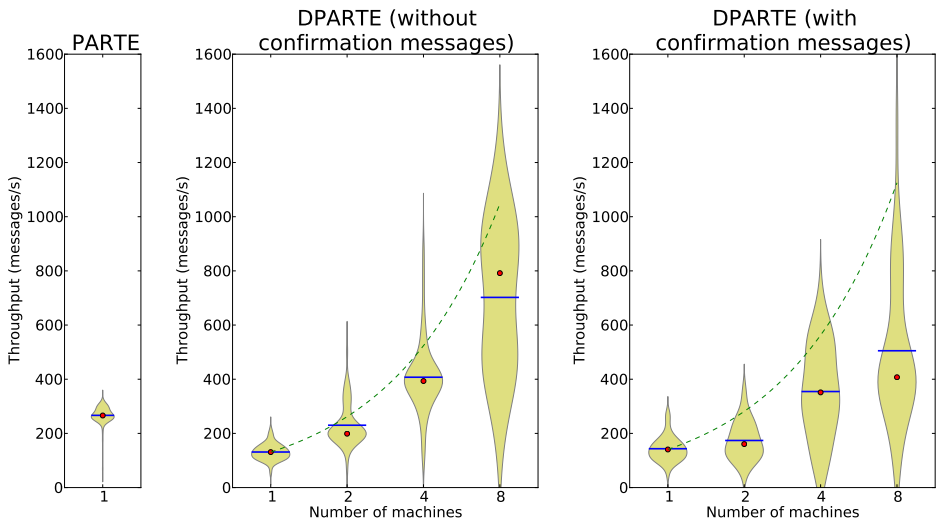


Figure 7.5: Throughput of PARTE, and DPARTE without and with confirmation messages, as the amount of machines increases. The green dashed lines indicate ideal speedup. Figure 7.6 contains a table of these results.

Throughput (msg/s)	1	2	4	8
PARTE	266 (1.00x)	—	—	—
DPARTE (no conf.)	131 (0.49x)	199 (0.75x)	393 (1.48x)	792 (2.98x)
DPARTE (conf.)	141 (0.53x)	161 (0.60x)	351 (1.32x)	407 (1.53x)

Figure 7.6: Median throughput of PARTE, and DPARTE without and with confirmation messages, as the amount of machines increases; and ratios compared to PARTE on one machine.

initial rule (a) and final rule (c) are on the 16$^{\text{th}}$ machine. In case of eight machines, the setup of 16 machines is changed so that sets of two machines are merged into one; similarly to move to four machines the setup of eight machines is taken with every two machines replaced by one, etc.

7.3.2 Results

Figure 7.5 and Figure 7.6 show the results. The throughput of the system increases as the amount of machines increases, but the speedup is less than an ideal speedup. The system with confirmation messages performs, as can be expected, less well than one without confirmation messages: using eight machines the throughput of the variant with confirmation messages is only slightly more than half of the throughput of the variant without confirmation messages (407 vs. 792). Note that in case only one

machine is used, both variants of DPARTE are equivalent, as confirmation messages are only sent for remote communication.

Compared to PARTE, DPARTE on one machine performs worse (as also seen in the previous experiment), the same is true when using two machines (speedup of 0.75 without confirmation messages). Going from one machine to two does not provide much benefit for this benchmark (this is especially clear in the case of the system with confirmation messages): this is because in the case of one machine all communication is local while when moving to two machines (de)serialization and network communication is necessary.

When further increasing the number of machines, a speed-up is achieved. Using eight machines for the system without confirmation messages increases the throughput to 792 messages per second, which is almost three times the throughput of PARTE (266 messages per second) and six times the throughput of the system with only one machine (131 messages per second).

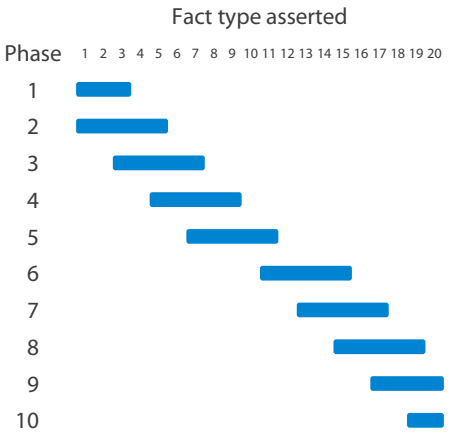
7.4 Mobile actors: benefit of dynamic load balancing and elasticity

7.4.1 Setup

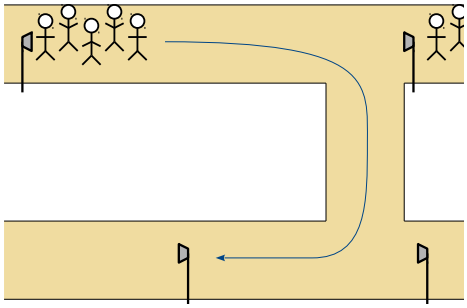
This final benchmark aims to prove the benefits of dynamic load balancing and elasticity. For this, three scenarios are compared: one in which the Rete network is manually distributed over two machines and dynamic load balancing is disabled (i.e. the same nodes stay on the same machine during the whole execution of the program), secondly a scenario that starts from the same configuration but allows dynamic load balancing (nodes can move between the two machines at run time), and lastly a configuration in which an empty third machine is present (while running, the Rete network will be redistributed and use the third machine). The second scenario shows the benefit of dynamic load balancing, the third shows the elasticity provided by the system.

As Figure 7.7 shows, this benchmark aims to simulate a scenario in which e.g. a crowd or a traffic jam moves, each time being within the view of several cameras, and whenever it moves out of view of one camera it enters the view of another. In the simulation, 10 phases of 1000 facts are asserted, each phase consisting of facts of three to five types, and between each assertion there is a pause of 0.5 ms. So while in the previous tests, all facts in the benchmark were asserted at once and the experiment measured the time until all of them were processed, here the facts are asserted over a period of five seconds (the time until all of them are processed is still measured).

The Rete network for this benchmark (Figure 7.8) consist of 20 rules (one for each



(a) Partitioning of benchmark into phases: 20 fact types are asserted in 10 phases, between three and five fact types are asserted in a phase, gradually progressing through them. 1000 facts are asserted per phase, so 10 000 facts are asserted in total.



(b) A scenario that could lead to such a benchmark. A crowd moves through an area, within view of multiple cameras. As they move, they go out of view of one camera and move into the range of another.

Figure 7.7: "20 rules in 10 phases" benchmark setup and motivation.

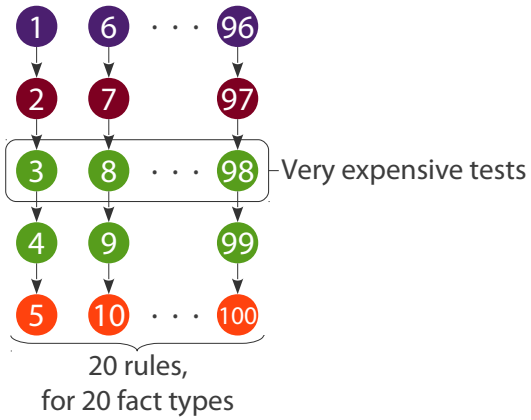


Figure 7.8: Rete network for "20 nodes in 10 phases" benchmark. There are 20 rules for the 20 fact types in the benchmark, each rule consists of: 1) a constant-test node that acts as the entry node for that fact type, 2) an adapter node that binds the timestamp of the fact to the variable ?t, 3) a beta-test node with a computationally expensive test, 4) a beta-test node that tests ?t > 95000, 5) a terminal node that ends the benchmark.

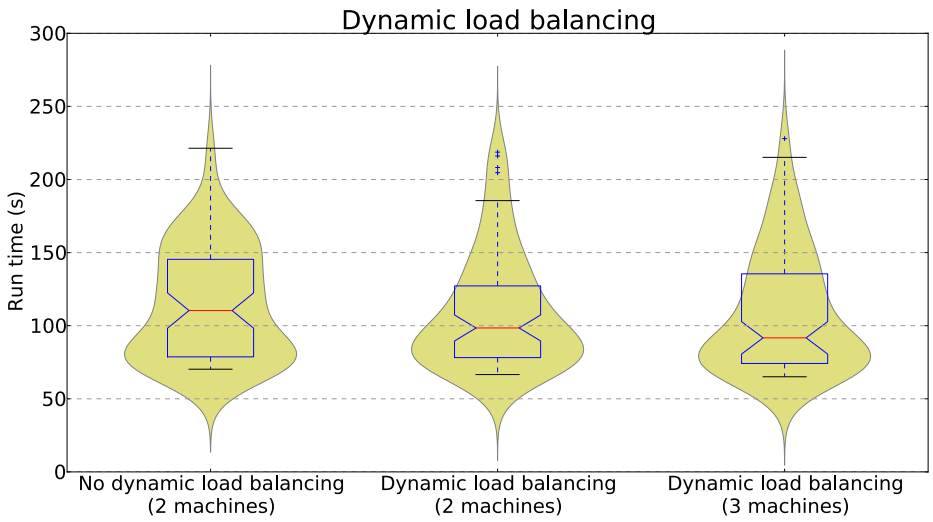


Figure 7.9: Comparison of system with manual distribution of the Rete network over 2 machines and (a) no load balancing, (b) load balancing, (c) load balancing and an extra third machine. The blue +s are outliers. The median result for (a) is 110.36 seconds; (b) 98.45 s (-10.8%); (c) 91.69 s (-16.9% compared to (a)).

fact type), each of them contains one computationally expensive node. The rule for the 20th fact type will trigger the end of the benchmark when it receives the last fact.

Every configuration starts from a manual partitioning of the Rete network over two machines. In this partitioning, the nodes for the first ten rules are put on machine one, the nodes for the other ten rules on machine two. Since in a real-life scenario one might not know what the asserted data will look like, this is a reasonable partitioning.

Figure 7.9 compares the three cases. Confirmation messages were disabled for this experiment.

7.4.2 Results: dynamic load balancing

Ideally, dynamic load balancing will see that, during the first half of the program, the first machine is overloaded, and move some of its nodes to the second machine. During the second half of the program, ideally some nodes move to the first machine.

Comparing the first two cases in Figure 7.9, after enabling dynamic load balancing the run time of the benchmark generally decreases. Compared to the configuration without dynamic load balancing, the bulk of the results have a lower run time, and the median has decreased (from 110.36 seconds to 98.45, so -10.8%).

Run	Nodes moving from 1 to 2	Nodes moving from 2 to 1	Run time (s)
1	28, 23	23, 93	80.2
2	8, 28, 9, 18	—	152.0
3	18, 3	—	112.7
4	3, 38	38, 98, 63, 68, 83, 53	99.2
5	43, 33	—	75.0
6	48	—	75.3

Figure 7.10: Movement of nodes in some typical runs of the program. See Figure 7.8 for the Rete network corresponding to this benchmark. The third node in every rule, i.e. nodes with IDs ending in 3 or 8, are computationally expensive nodes, so they are most worthwhile to move. Nodes ≤ 50 are originally on the first machine, nodes > 50 are originally on the second machine.

By examining the log files generated by these benchmarks in more detail, as shown in Figure 7.10, it is indeed the case that in the first half of the program nodes belonging to the rules triggered at that moment move from the first machine (which is overloaded) to the second machine (which is underloaded). In the second half nodes might move from machine two to one, although this is not always the case, as even though machine two is overloaded, machine one might still be overloaded as well.

Even though in most cases dynamic load balancing results in lower run times, there are some outliers that have a higher run time than without load balancing. Examining the log files generated by those outliers shows that there are some cases where the load balancing algorithm makes bad decisions. It for example decides to move a node from the first to the second machine right before half of the phases have completed, so just after the movement is completed machine two starts to become increasingly loaded while machine one is less loaded, and a node has to move back from machine two to one.

7.4.3 Results: elasticity

The third configuration in Figure 7.9 is a situation in which all nodes are manually distributed over two machines, as in the previous cases, but an (initially empty) third machine is available. Ideally, the network is redistributed at run time so that the nodes are spread (more or less) evenly over the three machines; this is elasticity.

The results show that this configuration again leads to a decreased median result compared to the scenario with only two machines (from 110.36 s without load balancing to 98.45 s with load balancing on two machines to 91.69 s for load balancing on three machines). Again, there are outliers when the load balancing algorithm makes a bad decision.

The log files indicate that, as desired, in the first half of the program nodes move from machine one to three, and in the second half nodes move from machine two to three.

7.4.4 Conclusion

From these results, it can be concluded that both dynamic load balancing and elasticity provide a benefit: the performance of the system generally increases after load balancing was added, and adding another machine causes the run time to decrease more. However, there are some cases in which dynamic load balancing decreases the performance, caused by the load balancing algorithm making bad decisions. The algorithm implemented here is simplistic, and could definitely be improved (subsection 8.2.1).

7.5 Conclusions

From these experiments, the following conclusions can be drawn:

- Communication is expensive, both locally (within a machine) and remotely (between machines). Computationally expensive use cases scale better than cases with a lot of communication.
- Adding more machines increases the throughput of the system for the given benchmarks, although the speedup achieved is less than ideal. Rete networks with many computationally expensive nodes will scale better, it needs to be possible to partition the Rete network in such a way that the communication overhead is small.

  For the tested benchmark, using eight machines the throughput of DPARTE (without confirmation messages) is three times as much as it was for PARTE on one machine, and six times as much as it was for DPARTE on one machine.
- The system with confirmation messages scales less well than one without confirmation messages. In the experiment, using eight machines, enabling confirmation messages (almost) halved the throughput. Confirmation messages should be disabled unless absolutely necessary.
- Dynamic load balancing increases the performance of the system in most cases where there is an overloaded machine. In the tested benchmark, the median run time decreased with 10.8% after enabling load balancing.

  There are a few cases in which load balancing decreases the performance, as a result of bad decisions by the load balancing algorithm. This is because the load balancing algorithm is too simplistic, it could be improved to reduce these cases.

- The load balancer can provide elasticity: if a machine is added at run time to a Rete network with overloaded machines, it is possible to repartition the Rete network to increase the performance. In the experiment, adding a third machine to a use case with two overloaded machines decreased the run time with 16.9% compared to a situation with only two machines and no load balancing.

Chapter 8
Conclusion

8.1 Summary and contributions

Many applications, such as traffic monitoring, crowd management or gesture recognition systems, aim to recognize patterns in streams of data, captured by a multitude of sensors. A good approach is to use rules: using a declarative language the programmer defines rules that describe patterns. As both hardware advances and applications want to recognize more complex patterns, the amount of data and the amount of rules increases, and existing solutions for such systems are either limited to a single machine or require manual programming to scale.

In this thesis, DPARTE, a system for large-scale pattern recognition on distributed architectures was created. It builds on PARTE, which uses the Rete algorithm to convert the rules into a network of nodes, and represents each node by an actor. DPARTE adds support for using multiple machines by allowing its actors to be distributed over these machines. Furthermore, its actors are mobile: their code, internal state and execution state can move from one machine to another. A load balancer gathers information on the load of all machines and decides when an actor should move from a certain machine to another.

We test the hypotheses that, 1) spreading the nodes of the Rete network over multiple machines causes the processing power requirements and memory usage to be divided amongst these machines, which should lead to an increased throughput, and 2) dynamic load balancing can be added to the system using mobile actors, and this provides an additional increase in performance.

Experimental evaluation showed that:

- This system is scalable: using an increasing amount of machines increases the throughput. For the tested benchmark, using eight machines the throughput of DPARTE (without confirmation messages) is six times as much as it is using one machine, and three times as much as it is for PARTE on one machine.

- It is elastic: if the amount of work increases or decreases during the execution of a program, it is possible to add or remove machines and repartition the Rete network. In an experiment, adding a third machine to a use case with two overloaded machines decreases the run time with 16.9% compared to a situation with only two machines and no load balancing.
- The system can handle variations in the type of facts asserted into the system or the rules triggered, through dynamic load balancing. Dynamic load balancing can thus increase performance. In the tested benchmark that aims to simulate a situation where cameras monitor crowds, and therefore rules are triggered in phases, the median run time decreases with 10.8% after enabling load balancing.

There are some limitations to the system, such as its large communication overheads (especially for remote communication), the fact that the initial partitioning of the Rete network still has to happen manually, and that the load balancing algorithm in the current implementation is too simple and as a result might make bad decisions. These limitations could be improved upon in future versions.

To our knowledge, DPARTE is the first Rete-based rule engine with a unified and distributed fact base, that supports automatic distribution and dynamic load balancing.

8.2 Possible directions for future research

This section discusses some improvements that could be made to the current system, and some open questions.

8.2.1 Smarter algorithm for automatic distribution and load balancing

Currently, DPARTE relies on the programmer to specify the initial distribution of the Rete network manually, after which a simple load balancing algorithm (subsection 6.2.2) will move overloaded nodes based on their inbox length.

First of all, an algorithm could be added to do the initial partitioning of the Rete network automatically, so it is not necessary for the programmer to know what the Rete network looks like and for him to determine in which way to best partition it. How to design this algorithm is still an open question.

Secondly, the load balancing algorithm that is currently implemented is very simple: it takes into account only the inbox lengths of the actors and uses some very simple heuristics to determine whether nodes should move and how. This algorithm could be extended to take into account other variables, such as processor, memory and network usage, the connectedness of nodes in the Rete network, and a more extensive profitability estimation to determine when and where a move should happen.

8.2.2 Granularity of the parallelism

In the current implementation, just as in PARTE, every Rete node is represented by one actor. However, this one-to-one mapping might not always be the best.

In some cases, finer parallelism could be possible: in join nodes especially, intra-node parallelism could speed up the matching of incoming tokens to the tokens in the memories of the node.

In other cases, coarser granularity is desirable. For example, in a Rete network that contains a long series of nodes with short execution time, the communication overhead is too large compared to the actual amount of computation happening (see for example the "Simple Tests" benchmark in section 7.2). Such a series of nodes could be merged into a single actor to avoid the communication overhead.

Furthermore, some nodes could be duplicated to allow for more parallelism. Test nodes for instance, do not contain any internal state, it is therefore no problem to clone such a node so that half of the tokens are processed by the original and half by the clone. The clone could then be put on another machine, if this avoids costly network communication.

It would even be possible to merge and split actors at run time, allowing the system to respond to variations in the work while the program is running and making it possible for the algorithm to use information that is only available at run time (such as how often certain communication paths are used or how much processing power a node uses on average). How this could best be done, and what considerations to take into account when designing such an algorithm, remains an open question.

8.2.3 Fault-tolerance

In this thesis, no particular attention was paid to making the system fault-tolerant. In DPARTE, if one machine crashes the whole system will fail. It was reasoned that the overhead of adding fault-tolerance would be unacceptable.

An open research question remains how to add fault-tolerance to a distributed Rete engine, and whether it is even possible to do so without causing a too large overhead.

Bibliography

[1] G. A. Agha. *Actors: a model of concurrent computation in distributed systems*. PhD thesis, Massachusetts Institute of Technology, 1985.

[2] R. Andrews, J. Diederich, and A. B. Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems*, 8(6):373–389, Dec. 1995.

[3] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. A Rule-Based Language for Complex Event Processing and Reasoning. *Web Reasoning and Rule Systems*, 6333:42–57, 2010.

[4] M. M. Aref and M. A. Tayyib. Lana–Match algorithm: a parallel version of the Rete–Match algorithm. *Parallel Computing*, 24(5-6):763–775, June 1998.

[5] D. Batory. The LEAPS algorithms. Technical report, University of Texas at Austin, 1994.

[6] J. M. Benitez, J. L. Castro, and I. Requena. Are artificial neural networks black boxes? *IEEE transactions on neural networks*, 8(5):1156–64, Jan. 1997.

[7] A. E. Berlonghi. Understanding and planning for different spectator crowds. *Safety Science*, 18(4):239–247, Feb. 1995.

[8] W. F. Clocksin and C. S. Mellish. *Programming in PROLOG*. Springer Verlag, 2003.

[9] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, Oct. 1989.

[10] W. De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. Phd thesis, Vrije Universiteit Brussel, 2004.

[11] P. Doherty and P. Rudol. A UAV search and rescue scenario with human body detection and geolocalization. In *Proceedings of the 20th Australian joint con-*

ference on Advances in artificial intelligence, Lecture Notes in Computer Science, pages 1–13, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[13] E. Friedman-Hill. Jess, the Rule Engine for the Java Platform, 2013.

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

[15] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57, Oct. 2007.

[16] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel algorithms and architectures for rule-based systems. *ACM SIGARCH Computer Architecture News*, 14(2):28–37, June 1986.

[17] J. L. Hintze and R. D. Nelson. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician*, 52(2):181–184, May 1998.

[18] L. Hoste. *Aligning Programming Paradigms with the Multi-Touch Revolution*. Master thesis, Vrije Universiteit Brussel, 2010.

[19] L. Hoste, B. Dumas, and B. Signer. Mudra: a unified multimodal interaction framework. In *Proceedings of the 13th international conference on multimodal interfaces - ICMI '11*, pages 97–104, New York, New York, USA, 2011. ACM Press.

[20] W. Hu, T. Tan, L. Wang, and S. Maybank. A Survey on Visual Surveillance of Object Motion and Behaviors. *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, 34(3):334–352, Aug. 2004.

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

[22] A. Jain and P. Duin. Statistical pattern recognition: a review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4–37, 2000.

[23] JBoss. Drools - The Business Logic integration Platform, 2013.

[24] P. Kampstra. Beanplot: a boxplot alternative for visual comparison of distributions. 2008.

[25] 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.

[26] J. Laird. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, Sept. 1987.

[27] 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.

[28] S.-H. Liao. Expert system methodologies and applications—a decade review from 1995 to 2004. *Expert Systems with Applications*, 28(1):93–103, Jan. 2005.

[29] H. Lü and Y. Li. Gesture coder: a tool for programming multi-touch gestures by demonstration. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12*, page 2875, New York, New York, USA, 2012. ACM Press.

[30] D. Luckham. *The power of events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. 2002.

[31] S. Marr, T. Renaux, L. Hoste, and W. De Meuter. Parallel Gesture Recognition with Soft Real-Time Guarantees. 2013.

[32] J. McDermott, A. Newell, and J. Moore. The efficiency of certain production system implementations. *ACM SIGART Bulletin*, (63):38, June 1977.

[33] D. Miranker and B. Lofaso. The organization and performance of a TREAT-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3–10, Mar. 1991.

[34] D. P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems; Long Version. Technical report, 1987.

[35] S. Mitra and T. Acharya. Gesture Recognition: A Survey. *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, 37(3): 311–324, May 2007.

[36] T. Renaux. *Parallel Gesture Recognition with Soft Real-Time Guarantees*. Master thesis, Vrije Universiteit Brussel, 2012.

[37] 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 - AGERE! '12*, page 35, New York, New York, USA, 2012. ACM Press.

[38] G. Riley. CLIPS: A Tool for Building Expert Systems, 2013.

[39] C. Scholliers and L. Hoste. Midas: a declarative multi-touch interaction framework. In *TEI '11 Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, pages 49–56, 2011.

[40] J. Sime. Crowd psychology and engineering. *Safety Science*, 21(1):1–14, Nov. 1995.

[41] E. A. Suma, B. Lange, A. S. Rizzo, D. M. Krum, and M. Bolas. FAAST: The Flexible Action and Articulated Skeleton Toolkit. In *2011 IEEE Virtual Reality Conference*, pages 247–248. IEEE, IEEE, Mar. 2011.

[42] M. Tambe, D. Kalp, A. Gupta, C. Forgy, B. Milnes, and A. Newell. Soar/PSM-E: investigating match parallelism in a learning production sytsem. In *Pro-*

ceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems - PPEALS '88, volume 23, pages 146–160, New York, New York, USA, 1988. ACM Press.

[43] A. S. Tanenbaum and M. Van Steen. *Distributed systems: Principles and Paradigms*. Pearson Prentice Hall, Upper Saddle River, NJ, second edition, 2006.

[44] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*. Academic Press, fourth edition, 2009.

[45] F. Wang, S. Liu, P. Liu, and Y. Bai. Bridging physical and virtual worlds: complex event processing for RFID data streams. *Advances in Database Technology - EDBT*, 3896(2006):588–607, 2006.

[46] F. Wang, S. Liu, and P. Liu. Complex RFID event processing. *The VLDB Journal*, 18(4):913–931, Mar. 2009.

[47] J. Watts and S. Taylor. A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):235–248, Mar. 1998.

[48] M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.

[49] A. Wilson and A. Bobick. Parametric hidden Markov models for gesture recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(9): 884–900, 1999.

[50] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, page 407, New York, New York, USA, 2006. ACM Press.

[51] 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.