# Distributed Prolog Reasoning in the Cloud for Machine-2-Machine Interaction Inference

**Radovan Zvoncek**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

### Examination Committee

Chairperson: Prof. Pedro Manuel Moreira Vaz Antunes de Sousa
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Johan Montelius

**June 2013**

# European Master in Distributed Computing

This thesis is part of the curricula of the European Master in Distributed Computing (EMDC), a joint program among Royal Institute of Technology, Sweden (KTH), Universitat Politecnica de Catalunya, Spain (UPC), and Instituto Superior Tecnico, Portugal (IST) supported by the European Community via the Erasmus Mundus program. My track in this program has been as follows:

- First and second semester of studies: IST
- The third semester of studies: KTH
- The fourth semester of studies (thesis): IST (officially), internship at Ericsson Research, Stockholm, Sweden

# Acknowledgements

First and foremost I would like to express my gratitude to professor Luis Veiga who agreed to undertake the role of my supervisor for this thesis, provided me with enormous help, guidance and support throughout my work and stood by me until the very end.

Then I would like to thank professor Johan Montelius and professor Leandro Navarro for their support during my whole enrolment in the Erasmus Mundus European Master in Distributed Computing programme.

I would also like to thank my supervisors at Ericsson Research Joerg Niemoeller and Leonid Mokrushin for sharing with me their knowledge and experience whenever I approached them.

Next, I would like to use this opportunity to thank my friends and fellow students in EMDC who shared with me all the joys and sorrows of studentship.

Last but not least I want to thank my family for the support and faith they have been putting in me during my whole academic career.

Lisboa, June 2013

Radovan Zvoncek

Dedicated to my mother Alena.

# Resumo

A interligação a nível aplicacional de vastos números de dispositivos não é uma tarefa trivial devido aos múltiplos protocolos e semânticas usados. O desenho de protocolos de tradução e de proxies mitigou o problema mas não oferece uma solução completa. O middleware baseado em ontologias tem o potencial de completar esta omissão.

Neste trabalho propomos o desenho e implementação de um middleware baseado em ontologia destinado a aplicação em larga escala. O nosso desenho estende o motor de inferência convencional Prolog, particionando a sua base de dados através de uma DHT, e realizando a migração do contexto de avaliação de objectivos entre diferentes instâncias do motor. Assim, consegue-se elevada paralelismo através de um modelo de descentralizado de computação cooperativa.

Os testes realizados demonstram que as penalizações introduzidas são largamente compensadas quando o volume de carga imposto ao sistema é elevado.

# Abstract

Application layer interconnection of vast amounts of devices is not a trivial task due to numerous protocols and semantics devices use. Designing translating protocols and proxies mitigated the problem but did not provide a complete solution. Ontology-based middlewares have the potential to complement the gap.

In this work we propose the design and implementation of an ontology-based middleware aiming for massive-scale deployment. Our design extends a conventional Prolog engine by sharding its database using a DHT and consequently migrating the goal evaluation context among different engine instances, thus achieving decentralised and collaborative computation model offering high degree of concurrency.

Experiments performed with our implementation show that overhead introduced by managing a distributed system is compensated once the system load is sufficiently high.

# Palavras Chave
# Keywords

## Palavras Chave

Arquitecturas descentralizadas

Computação cooperativa

Inferência distribuída

Middleware

Ontologias

## Keywords

Collaborative computation

Decentralised architecture

Distributed inference

Middleware

Ontology

# Index

# List of Figures

# List of Tables

# Acronyms

**DHT** Distributed Hash Table

**EMDC** European Master in Distributed Computing

**FOL** First Order Logic

**FOPL** Frst Order Predicate Logic

**GMP** Generalised Modus Ponens

**MP** Modus Ponens

**KB** Knowledge Base

**OWL** Web Ontology Language

**P2P** Peer-to-Peer

**SLD** Selective Linear Definite

**RDF** Resource Description Framework

# 1 Introduction

The vision of invisible or ubiquitous computing (Weiser 1993) of enabling Internet connectivity to every device that can potentially benefit from being connected has been almost fulfilled. The number of devices connected to the Internet has reached 10 billion and is expected to grow (Ericsson 2011). However, providing connectivity to devices without applications that can utilise it is not sufficient to unlock the full potential of a globally interconnected infrastructure.

While many of the obstacles in connecting the devices have been successfully solved, facilitating the actual communication keeps facing several problems. Devices are manufactured by different vendors and therefore use vendor-specific, non-standardized protocols and semantics. Consequently, devices using the same protocol families become integrated and tightly coupled in vertical domains, thus effectively disabling any generic interconnection initiative. As a result, the devices literally do not understand each other.

Although there has been significant work done regarding designing new protocols and implementing translating proxies aiming to bridge the gaps between different vertical domains (Schilit & Sengupta 2004), there is at least one more complementary approach. Giving the applications an ability to reason, learn and interpret observed behaviours in their environment, based on contextual knowledge, would facilitate horizontal communication without the need of detailed manual integration of disjoint vertical domains.

However, there is no clear answer on how this ability should be introduced. Imposing additional computation on the devices themselves is not feasible as they already often strive to limit resource consumption. On the other hand, the vast number of devices implies strict performance requirements on systems used to act as middleware attempting to off-load the computation from the devices.

Previous work done by Ericsson Research resulted in creating the conceptual system of a platform attempting to develop a system enabling device interoperation on application level. The system is named Axon and Figure 1.1 illustrates its envisioned 3-layer architecture. In

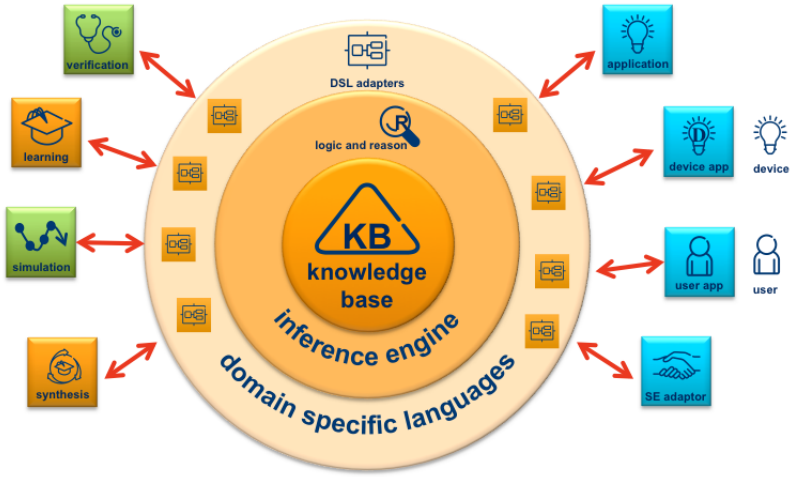



Figura 1.1: Illustration of a platform offering reasoning functionality.

its core, Axon contains the knowledge base gathering all the data available in the system (e.g. sensor readings), as well as additional information about how to utilise the data (e.g. rules and theories). The second layer is the inference engine consisting of Prolog core facilitating the actual reasoning about the available knowledge. The outer layer consists of adapters to domain specific languages for other components present in the system (e.g. applications, machine learning).

In this thesis, we will attempt to develop an implementation of Axon's knowledge base and reasoning engine.

## 1.1 Shortcoming of Current Solutions

A system facilitating the communication of autonomous devices at the application layer should meet the following requirements:

1. Information must be efficiently accessible for any kind of manipulation regardless its amount.
2. Interaction of devices must be interactive and happen in the order of few seconds.
3. Devices must not be burdened with excessive overhead.

The main aim of this thesis is to answer the following question:

*Is it possible to design and implement a distributed system, based on the concept of Axon, that would allow efficient data access and interactive operation without burdening the actual devices by additional overhead, while providing feasible back-end for facilitating reasoning for the devices?*

## 1.2 Contributions

Our goal is to find a design and implementation of a system that would demonstrate the feasibility of using a distributed middleware for device communication based on reasoning.

We will try to accomplish this by showing that the benefit gained from the sound design and implementation of a distributed middleware will compensate for the inevitable overhead introduced by distribution in a new system. The gained benefit can be manifested as the ability to handle more clients and bigger volume of information than a comparable sized centralised system.

The main contributions of this thesis are:

- Scalable design of a full-featured distributed Prolog engine.
- Decentralised middleware with the ability to sustain operation under a load excessive for a comparable centralised approach.
- Functioning prototype of to this day only conceptual System.
- Throughout evaluation of the proposed system including the assessment of correctness, scalability and performance.

In addition, the core ideas presented in this thesis have been partially described in a submission to Inforum 2013 titled "Distributed Prolog Reasoning for Cloud-based Machine-2-Machine Interaction Inference".

## 1.3 Structure of the Document

This dissertation is divided into 6 chapters. Chapter 1 serves as an introduction to this thesis. It explains the background for the problem addressed by this work and positions the work within

a broader context. It also formulates the main research question, the main contributions, and key results of this thesis.

The purpose of Chapter 2 is to introduce the underlying concepts this thesis builds on. It explains the concepts of knowledge and reasoning, describes how they are used for device interoperation, and overviews the existing systems addressing the problem of such functionality and their shortcomings from our point of view.

Chapter 3 presents the decisions made during the process of finding a solution for the aforementioned question. It is split into sub-chapters, each addressing a major problem that needed to be solved. The chapter fist provides general overview of the solution, then continues by introducing the approach taken to build a storage back-end and the description of the related computational model. Then, the chapter discusses how we approached the trade-off between consistency, availability and partition-tolerance.

Consequently, Chapter 4 describes the implementation details of the designed system. It explains the motivation behind the choice of tools used to implement particular aspects of the proposed system.

Chapter 5 then includes the evaluation of the implemented system. It first introduces the methodology used for the evaluation with a centralised system used as a reference, and continues by assessing key parameters of both deployments.

and then focuses on presenting the performance of the system with results describing observed overhead expected to be present in the system and then demonstrates the scalability and load distribution provided by the implemented system.

Finally, chapter 6 summarises this thesis and draws possible directions for future work.

# 2 Related Work

Axon is aiming to unlock the potential of interconnected devices by introducing concepts such as knowledge and reasoning, and linking them to logical programming and Prolog in particular. There is a huge gap between abstract concepts of knowledge and reasoning and actual implementation of Axon. This chapter will try to bridge this gap by explaining the aforementioned concepts and introducing links necessary to allow understanding and use in implementation.

Section 2.1 will briefly explore how have these concepts been defined in the literature and how do they relate to Prolog and to the aim of Axon. Next, section 2.2 describes Axon's proposed knowledge model and representation. Section 2.3 will shortly recapitulate the concepts of distributed hash tables. Section 2.4 will present existing systems related to Axon.

## 2.1 Knowledge and Reasoning

Finding a definition of "knowledge"has taken countless philosophers countless years and still has not reached its conclusion. The meanings of other terms are comparably dim. Regardless the absence of formal definitions, it is still possible to find at least informal explanations.

Brachman et al. (Brachman & Levesque 2004) provide the following explanation of knowledge, representation and reasoning:

- *Knowledge* is explained as a relation between a *knower* (i.e. an entity who possesses the knowledge) and a *proposition* (i.e. an idea expressed by a simple declarative sentence). Such relation is usually formulated as "knower knows proposition". However, the exact verb naming the relation can express any *attitude* (e.g. hopes, regrets, fears or doubts), which authors call *belief*. The propositions can be arbitrary declarative sentences but within the context of this work we will consider only the sentences that can be deemed true or false.

- *Representation* is explained as a relation of elements belonging to two different domains where the first can represent the second. The relation of representation is usually established in order to provide faster or easier manifestation of some concept from the second domain by its representation from the first domain.

- *Knowledge representation* is then an effort to use formal symbols to represent a set of propositions and beliefs.

- *Reasoning.* Since representation is not meant to express all believed propositions, it is the role of reasoning to (formally) manipulate the relations of representations and produce new knowledge.

When talking about knowledge, it is also necessary to introduce the term *ontology.* Russel et al. (Russell, Norvig, Canny, Malik, & Edwards 1995) define ontology as "[Ontology is a] particular theory of the nature of being or existence. The ontology represents all things that exists but does not say anything about their specific properties and interrelationships."This definition is compatible with the definition of knowledge provided by Brachman et al. in a sense that both knowledge and ontology provide a static description of a known world. The difference lies in the scope. While knowledge usually refers to all information present in the system, ontology can refer to a particular subset of the knowledge.

### 2.1.1 Knowledge in Computer Systems

Brachman et al. provide explanation and argue the benefits of knowledge representation in computer systems. The main claimed benefit is the ability to describe the system in a more appropriate level of detail when higher level of abstraction is useful. This is a valid benefit, as computer systems can feature complex internal states based on many variables and conditions. Representing such systems in a different level of abstraction can be more comprehensible for an external observer or human observer.

Computer systems containing knowledge representation are then called *knowledge-based systems* that contain symbolic representations and have the following two properties:

- External observers can understand the systems as standing for propositions. This means the behaviour of a knowledge-based system can be described by some higher-level goal or

aim.

- Knowledge-based systems behave according to the knowledge representation they contain. This means it is always possible to explain why has the system behaved in one way or another.

*Knowledge base* (KB) is then defined as the symbolic representation of a knowledge-based system. KB then constitutes the ontology of the system.

After establishing a definition of knowledge-based systems, Brachman et al. assess the benefits of keeping the KB external. Their concern is that including the KB into the system itself would eliminate the need to reason with it during the system's operation and therefore make the system operate faster. The downside is that this modification would turn knowledge-based system into expert systems (Jackson 1990) and effectively remove their ability to cope with unforeseen and open-ended tasks. Keeping the KB explicit is said to yield the following benefits:

- Easy addition of new functionality that depends on previous knowledge.
- Possibility to extend existing behaviour by adding new beliefs.
- Ability to analyse faulty behaviour by locating erroneous beliefs.
- Opportunity to fully explain and justify the behaviour of the system.

Another stated benefit of knowledge-based systems is the perseverance of the reasoning functionality. Reasoning allows a knowledge-based system to act based on what it believes (i.e. any outcomes of the preliminary reasoning in addition to knowledge explicitly stored in the KB). In turn, the KB itself can contain only simple and general facts that can be later (re)used in multiple situations.

Alternatively to the definition of knowledge, Russel et al. introduces the term *knowledge-based agents*. They positions them as the next step after problem-solving agents with limited knowledge embedded in their code, and agents using value-driven decisions of actions to take. Similar to Brachman, it introduces the term *knowledge-base* as a set of sentences, where *sentence* is an elementary unit of knowledge. He uses the term inference to name aforementioned reasoning. Finally, he distinguishes between declarative and procedural knowledge-based agents. The

prior initiate their operation with blank KBs and build them iteratively during their operation, while the latter contain desired behaviours as a priori available knowledge.

### 2.1.2 Logic

Formal logic systems can be used to link abstract concepts of knowledge and reasoning to practical systems such as Prolog. This section will explain how this can be achieved.

There are various kinds of formal logic (e.g. propositional, first-order, infinitary or intuitionistic logic). For the purpose of this thesis, we need to identify what are the characteristic features they have in common. Every logic defines:

- *Syntax* determining what sentences belong to the language of a given logic.
- *Semantics* of the sentences that determine the truth of the sentences with respect to each model.

In the context of formal logic systems, it is possible to define several other terms.

**Definition 1** (Model)**.** *A model is a mathematical abstraction that defines truth or falsehood of every sentence accepted by logic's syntax.*

**Definition 2** (Sentence Model)**.** *A sentence $\alpha$ is said to satisfy model $M$ when $\alpha$ is true in $M$. If sentence $\alpha$ satisfies model $M$, $M$ is said to be a model of $\alpha$. All models of $\alpha$ are denoted as $M(\alpha)$.*

**Definition 3** (Knowledge Base)**.** *Knowledge-base $KB_\alpha$ of a formal logic system is a language that contains sentences accepted by the syntax of of logic $\alpha$.*

**Definition 4** (Entailment)**.** *The relation of logical entailment is defined as*

$$\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$$

Informally, sentence $\alpha$ entails sentence $\beta$ if and only if in every model in which $\alpha$ is true, $\beta$ is also true.

**Definition 5** (Reasoning)**.** *Reasoning with knowledge base $KB_{\alpha\beta}$ is the process of establishing a relation of logical entailment between sentences $\alpha$ and $\beta$ where $\alpha, \beta \subseteq KB_{\alpha\beta}$.*

Reasoning with KBs yield sentences that are not necessarily present in the KB thus creating new knowledge and allowing drawing conclusions from the existing knowledge. This process is called logical inference.

**Definition 6** (Logical Infernece)**.** *Logical inference is a process of deriving conclusions from a KB denoted as*

$$KB \vdash_i \alpha$$

*and say inference algorithm $i$ derives $\alpha$ from $KB$.*

Inference algorithms can have the following properties:

- *Soundness.* An inference algorithm that derives only entailed sentences is called sound.
- *Completeness.* An inference is algorithm is complete if it can derive any sentence that is entailed.

Generic logic does not define any inference algorithm. Therefore we will provide examples of two particular logics. Propositional Logic is included as an example of the simplest practically usable logic and First Order Logic (FOL) due to its role as the underlying formal model of Prolog explained later in section 2.1.3.

#### 2.1.2.1 Propositional Logic

Propositional logic is defined by the following syntax (in BNF notation):

```
       Sentence → AtomicSentence
                | ComplexSentence
 AtomicSentence → Proposition
                | True
                | False
ComplexSentence → ¬ Sentence
                | Sentence Connective Sentence
     Connective → ⋀ | ⋁ | ⇒ | ⇔
```

Figura 2.1: The syntax of Propositional logic.

The semantics of propositional logic are defined in Figure 2.1.

Propositional Logic introduces several new concepts:

| P | Q | ¬ P | P $\bigwedge$ | P $\bigvee$ Q | P $\Rightarrow$ Q | P $\Leftrightarrow$ Q |
|---|---|---|---|---|---|---|
| False | True | True | False | False | True | True |
| False | True | True | False | True | True | False |
| True | False | False | False | True | False | False |
| True | True | False | True | True | True | True |

Tabela 2.1: Semantics of propositional logic.

- *Logical equivalence.* Two sentences $\alpha$ and $\beta$ are logically equivalent if they are true in the same set of models.
- *Validity.* A sentence is valid if it is *True* in all models. Valid sentence is called a *tautology.*
- *Satisfiability.* A sentence is satisfiable if it is *True* in at least one model.

Inference algorithms provided by the propositional logic are two. The first one is called *Modus Ponens* (MP) and is defined as

$$\frac{\alpha \Rightarrow \beta,\ \alpha}{\beta} \tag{2.1}$$

This means that given sentences $\alpha \Rightarrow \beta$ and $\alpha$, it is possible to infer $\beta$.

Modus tollens (MT) is an alternative inference rule equivalent to MP available in propositional logic. MT is defined as

$$\frac{\alpha \Rightarrow \beta,\ \neg\beta}{\neg\alpha}$$

meaning that given implication $\alpha \Rightarrow \beta$ and negation of $\beta$ it is possible to infer the negation of $\alpha$. MT can be converted to MP using the mechanism of logical transposition saying that antecedent ($\alpha$) and consequent ($\beta$) in a conditional statement ($\alpha \Rightarrow \beta$) can be switched provided both negated. For example

$$(\alpha \Rightarrow \beta) \Leftrightarrow (\neg\beta \Rightarrow \neg\alpha)$$

The second rule is called *And-Elimination* and is defined as

$$\frac{\alpha \bigwedge \beta}{\alpha} \tag{2.2}$$

meaning that given a conjunction of two sentences, any of the sentences can be inferred.

Showing the soundness of Modus Ponens and And-Elimination can be done by considering all possible values of $\alpha$ and $\beta$.

The main benefits of propositional logic is that its sentence are context-independent because their meaning does not depend on the context. However, propositional logic has limited expressive power because its models can contain only *facts*.

#### 2.1.2.2 First-Order Logic

```
  Sentence → Atom
           | Sentence Connective Sentence
           | Quantifier Variable, ··· Sentence
           | ¬ Sentence
      Atom → Predicate
           | Predicate(Term, ···)
           | Term = Term
      Term → Function(Term,···)
           | Constant
           | Variable
Connective → ⋀ | ⋁ | ⇒ | ⇔
Quantifier → ∀ | ∃
```

Figura 2.2: The grammar of First-Order Logic.

First-order logic is defined in Figure 2.2. To explain the semantics of the FOL syntax, it is necessary to introduce several new concepts.

- Objects represent real-world elements (e.g. squares, pits, legs).
- *Domain* is a non-empty set of objects associated to a model of FOL.
- *Constant* is a reference to a real-world object.
- *Relations* are tuples of objects that are somehow related in the real world. Relations are instantiated as predicates and functions.
- *Predicate* stands for a relation and returns *True* if its arguments are related in the given domain.
- *Function* is similar to predicate, but can return an arbitrary value.

- *Arity* is a number of arguments of a predicate or function.
- *Interpretation* is a specification determining which objects, relations and functions are referred to constants, predicates and function symbols
- *Quantifiers* allows expressing properties of collections of objects.
- *Variable* is a symbol that can represent any object.
- *Term* is a logical expression that refers to an object. A term can be either function, constant or variable. Term with no variables is called a *ground term.*

Inference in first-order logic can be achieved in two ways. The first one is based on converting the KB to propositional logic and then employing propositional inference defined in equations 2.1 and 2.2. Converting a FOL KB into a propositional KB can be achieved by two rules: Universal Instantiation and Existential Instantiation.

**Definition 7** (Universal Instantiation)**.** *Given variable $v$, sentence $\alpha$ and ground term $g$,*

$$\frac{\forall \nu\ \alpha}{SUBST(\{v/g\}, \alpha)} \tag{2.3}$$

The definition 7 means that we can infer any sentence obtained by substituting a ground term $g$ for the variable $v$. The notion of substitution is explained in greater detail in section 8.3 of (Russell, Norvig, Canny, Malik, & Edwards 1995).

**Definition 8** (Existential Instantiation)**.** *For any sentence $\alpha$, variable $v$ and constant symbol $k$ that does not appear elsewhere in the knowledge base,*

$$\frac{\exists v\ \alpha}{SUBST(\{v/k\}), \alpha} \tag{2.4}$$

The second approach (inferring sentences of FOL directly without converting them to propositional logic) is called *Generalized Modus Ponens* (GMP). The rule definition is as follows.

**Definition 9** (Generalised Modus Ponens)**.** *For atomic sentences $p_i, p_i\prime$, and $q$, where there is a substitution $\theta$ such that $SUBST(\theta, p_i\prime) = SUBST(\theta, p_i)$, for all $i$,*

$$\frac{p_1\prime, p_2\prime, \cdots, p_n\prime,\ (p_1 \bigwedge p_2 \bigwedge \cdots \bigwedge p_n \Rightarrow q)}{SUBST(\theta, q)} \tag{2.5}$$

To show the soundness of GMP we first observe that for any sentence $p$ and for any substitution $\theta$

$$p \vDash SUBST(\theta, p)$$

holds because of definition 7. It holds particularly for $\theta$ that satisfies the conditions of definition 9. Thus, from $p_1\prime, p_2\prime, \cdots, p_n\prime$ we can infer

$$SUBST(\theta, p_1\prime) \bigwedge \cdots \bigwedge SUBST(\theta, p_n\prime) \tag{2.6}$$

and from the implication $p_1 \wedge \cdots \bigwedge p_n \Rightarrow q$ we can infer

$$SUBST(\theta, p_1) \bigwedge \cdots \bigwedge SUBST(\theta, p_n) \Rightarrow SUBST(\theta, q). \tag{2.7}$$

Substitution $\theta$ in GMP is defined so that $SUBST(\theta, p_1\prime) = SUBST(\theta, p_1)$ for $\forall i$, therefore $p_i\prime$ exactly matches $p_i$. Hence, $SUBST(\theta, q)$ follows Modus Pones that we have shown to be sound.

The GMP rule requires finding substitutions that make different logical expression look identical. This process is called *unification* and is more closely explained in section 9.2.2 of (Russell, Norvig, Canny, Malik, & Edwards 1995).

### 2.1.3 Horn Clauses and Logical Programming

Inference algorithms for both propositional logic and FOL are powerful due to their provable soundness and completeness. However, the full power of resolution is not always needed in practice. Moreover, the computation complexity of these algorithms is often unacceptable.

To overcome these limitations, certain restrictions can be imposed on sentences contained in KBs. One such restriction is to contain only Horn clauses (Horn 1951), which are clauses that are disjunctions of literals of which exactly one is positive, for example

$$\neg A \bigvee \neg B \bigvee \cdots \bigvee \neg C \bigvee X \tag{2.8}$$

Restricting Horn clauses one step further towards less general case gives *definite clauses*, which are disjunctions of literals of which exactly one is positive. Definite clauses are used as the basis of Selective Linear Definite (SLD) resolution algorithm (Kowalski 1973) which is the

| A | B | C | $\neg A \bigvee \neg B \bigvee C$ | $(A \bigwedge B) \Rightarrow C$ |
|---|---|---|---|---|
| True | True | True | True | True |
| True | True | False | False | False |
| True | False | True | True | True |
| True | False | False | True | True |
| False | True | True | True | True |
| False | True | False | True | True |
| False | False | True | True | True |
| False | False | False | True | True |

Tabela 2.2: Logical equivalence of definite clauses and their representation as implication.

| FOL | Prolog |
|---|---|
| $(A \bigwedge B) \Rightarrow X$ | X :- A, B. |
| $True \Rightarrow X$ | X. |

Tabela 2.3: Expressing implications of FOL in Prolog.

fundamental concept of logical programming.

This brings the abstract concept of knowledge and reasoning only one step away possibility to be implemented by logical programming. The final step needed is to show how definite clauses can be transformed into equivalent form more understandable for humans and therefore more appropriate for describing knowledge.

Provided that the operation of disjunction is associative, Figure 2.2 proves that disjunctions of clauses can be also expressed by logically equivalent implications, for example

$$\neg A \bigvee \neg B \bigvee \cdots \bigvee C \bigvee X \Longleftrightarrow (A \bigwedge B \bigwedge \cdots \bigwedge C) \Rightarrow X \tag{2.9}$$

An implication in equation 2.9 is then understood as "*if A and B, and ... and C is True, then X is also True*". A and B is then referred to as *hypothesis* and C is a *conclusion.* A hypothesis can be also expressed as

$$True \Rightarrow C \tag{2.10}$$

what indicates that conclusion C is unconditionally true.

Implications with the syntax of equation 2.9 are considerably more practical when it comes to describing knowledge. In practice, they are most commonly expressed in the syntax of the Prolog programming language. Table 2.3 shows equivalent syntax for implications 2.9 and 2.10.

Prolog has been widely used for knowledge representation due to its equivalence with definite clauses and reasoning implemented by inferring new sentences using SLD algorithm.

### 2.1.4 Additional Terminology

Substantial part of this work extensively uses Prolog and its terminology. Here we will state the commonly used vocabulary in order to prevent possible misunderstandings. For the remainder of this thesis, we will use the following vocabulary:

- *Atom* is a literal with no inherent meaning.
- *Number* represents decimal or integral value.
- *Variable* is a string literal with first character capitalised. Variable can represent any term.
- *Term* is either an atom, number, variable or a compound term.
- *Compound term* consists of an atom called functor and a number of arguments that are again terms.
- *Clause* is Prolog notation of definite clause as defined in equation 2.8 and has the form of "Head :- Body".
- *Head* is equivalent to *conclusion*, e.g. X in equation 2.9.
- *Body* is equivalent to *hypothesis*, e.g. $(A \bigwedge B \bigwedge \cdots \bigwedge C)$ in equation 2.9.
- *Fact* is a unconditionally true *conclusion*, e.g. $C$ in equation 2.10,
- *Rule* is a clause with both hypothesis and conclusion.
- *Arity* is the number of arguments accepted by a term constituting a *head*.

## 2.2 Knowledge Representation and Reasoning in Axon

Previous work done regarding the Axon project has resulted in establishing a model for knowledge representation and reasoning enabling devices and applications handle contextual information and reason about their environment. This section will briefly present these concepts.

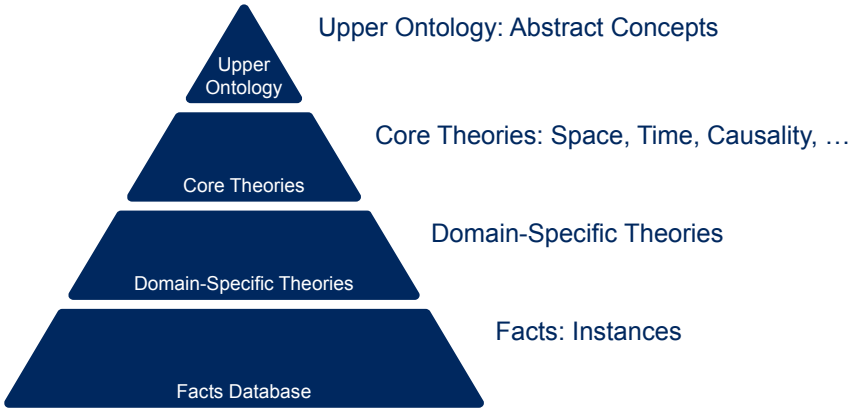



Figura 2.3: Hierarchical knowledge representation in Axon.

Axon assumes hierarchical structure of knowledge based on CYC (Matuszek, Cabral, Witbrock, & DeOliveira 2006) illustrated in figure 2.3. On its top, Axon's knowledge base contains the *Upper Ontology*. Upper Ontology contains abstract concepts that do not say much about the world at all. Instead, it represents very general relations between very general concepts. For example, it contains the assertions declaring that every event is a temporal thing and that every temporal thing is a thing. Thing is the most general concept. Everything whatsoever in the system is an instance of thing.

The second uppermost component contains *core theories* representing general facts about space, time and causality. For example, core theories would contain generic expression of the fact that if event $a$ causes event $b$, that $a$ precedes $b$ in time. Core theories are the essential theories to almost all common-sense reasoning.

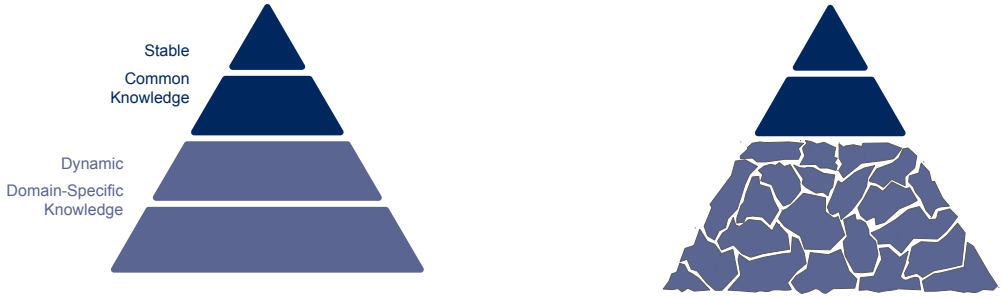
The third level contains *Domain-Specific Theories* that are more specific than core theories. Domain-specific theories apply to special areas of interest like the propagation of disease, finances, chemistry, etc. They are the theories that make the knowledge base practically useful, but are not necessary for common sense reasoning. For example, a domain-specific theory can say that for any person $p$ and any truck $t$, $p$ driving $t$ for more than 8 hours in a row is illegal and unsafe.

The bottom level is the *Facts Database* that contains ground-level facts. These are the statements about particular individuals in the world. Generalizations do not belong to this level

any more, as they should be placed in the Domain-Specific Theories. In contrast, this level contains anything that can be written as a newspaper headline, was obtained as a reading from a sensor or is a statement about particular individual in the world. For example, Facts Database would contain facts saying John is a person driving a truck $t$ and $t$ is loaded with 10 tons of rubber ducks.

Theories in general are sets of assertions (i.e. facts and rules) valid in a specific context. They can bundle assertions according to several aspects:

- Shared set of assumption on which the assertions' truth depends on.
- Shared topic (e.g. truck driving theory modelling the activity of driving a truck).
- Shared source (e.g. truck temperature theory representing temperature of various parts of a truck).




(a) Static and dynamic knowledge in Axon.

(b) Modular representation of dynamic knowledge in Axon.

Figura 2.4: Decomposition of knowledge in Axon.

The hierarchical organisation of knowledge allows dividing the knowledge-base into two parts as it is illustrated in Figure 2.4(a):

- Common knowledge composed of the two upper levels. This is the knowledge that is useful for any application domain and should therefore be always available. This knowledge is stable and does not change due to system operation. For example, a truck simply can not redefine the concept of causality.
- Domain-Specific knowledge is dynamic and contains knowledge specific to local domains (i.e. instances) of the knowledge base. It contains:

- Theories supplied by applications and devices themselves.
- Facts injected by application providers, device vendors and user themselves.
- Theories learned during the operation in the local domain.
- Facts learned from data generated in live operation, including user behaviour.

The proposed hierarchical structure is beneficial for a multiple of reasons.

At first, the stable parts of the knowledge base are comparably smaller than the dynamic ones. This is good particularly for distribution purposes as the resulting system is kept rather compact. Furthermore, independence of the upper-ontology and core theories from the domain-specific knowledge allows Axon to be applied in various domains.

The second benefit results from the possibility to organise the domain-specific knowledge in a modular way as it is described in Figure 2.4(b). Different modules of knowledge can be loaded depending on the given application domain and the devices devices actually present in the system.

## 2.3 Distributed Hash Tables

Distributed hash tables (DHTs) originated from the research in the area of distributed caching by Karger et al. (Karger, Lehman, Leighton, Panigrahy, Levine, & Lewin 1997). In their principle, they combine the following concepts:

*Tables.* Tables are conventional data structures implementing efficient lookup functionality. Tables can be seen as collections of entries, where each entry is assigned some value. This value is know as *key*. If the key is known, the entry can be efficiently added or retrieved from the collection. The main benefit of tables is that the manipulation with entries has constant ($O(1)$) complexity because it does not require any manipulation (e.g. search) with other entries present in the table. Depending on particular implementation, one key can be assigned to multiple entries, or one entry can be placed under multiple keys.

*Hashing.* Hashing (Knuth 2006) algorithms are functions transforming input of arbitrary and variable length into output of fixed length. The domains of functional values of hash functions are typically smaller that the domains of input values, which opens the possibility of

*collisions*, i.e. multiple input values being transformed into the same functional value. Hashing functions are typically easily computable, but it is extremely difficult to find an inverse function to a hashing function. Hashing functions can be used to obtain keys for entries stored in tables.

Partitioning a table according to some hash function then results in DHT. A DHT is a collection similar to a table, but the entries are clustered according to the provided hash function. In practice, this concept is typically implemented as a distributed system featuring multiple nodes, each responsible for managing disjoint cluster of entries.

DHTs employ a particular kind of hashing called *consistent hashing* first introduced by Karger et al. Consistent hashing allows changing the number of nodes participating in a DHT without the need to re-distribute all entries stored already in the DHT.

The original implementations of DHTs are Chord (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001), CAN (Ratnasamy, Francis, Handley, Karp, & Shenker 2001), Tapestry (Zhao, Kubiatowicz, Joseph, et al. 2001) and Pastry (Rowstron & Druschel 2001). Their comparison is summarized in figure 2.4. $N$ represents number of nodes in the system, $B$ represents number base for node identifiers.

Further comparison of different DHT implementations can be found in greater detail in (Wang & Li 2003) or (Lua, Crowcroft, Pias, Sharma, & Lim 2005).

| | Architecture | Lookup | Routing Performance |
|---|---|---|---|
| Chord | Uni-directional circular space | Hashing a key determines the node responsible for the entry | $O(logN)$ |
| CAN | Multidimensional space | Hashing key(s) determines coordinates in multi-dimensional space | $O(d.N^{1/d})$ |
| Pastry | Global mesh network | Matching output of the hashing function with the prefix of node identifiers | $O(\log_B N)$ |
| Tapestry | Global mesh network | matching the output of the hashing function with the suffix of node identifiers | $O(\log_B N)$ |

Tabela 2.4: Comparison of different DHT protocols.

## 2.4 State of the Art

Even though the work presented in this paper merges concepts of logical programming and distributed computing, it is not addressing any of the issues of parallel execution of Prolog programs, such as the ones surveyed in (Gupta, Pontelli, Ali, Carlsson, & Hermenegildo 2001). Our aim is to provide a support for many concurrent executions of a Prolog programs.

It is also not in the scope of this work to design a full-featured middleware for device interconnection such as the ones surveyed and described in (Teixeira, Hachem, Issarny, & Georgantas 2011) and (Bandyopadhyay, Sengupta, Maiti, & Dutta 2011). It is not our aim to deal with issues such as device discovery but focus solely on the application layer.

More closely related to our work is the effort to build semantic middlewares. Semantic middlewares facilitate device communication by allowing explicitly expressing the *meaning* of information devices exchange. In other words, the devices are allowed to *know* what given information means. Examples of semantic middlewares can be found in (Song, Cárdenas, & Masuoka 2010) and (Gómez-Goiri & López-De-Ipiña 2010).

In (Song, Cárdenas, & Masuoka 2010), the authors pick Semantic Web technologies to implement a layer of abstraction above heterogeneous devices and thus providing interoperability at the application layer. However, the available evaluation is limited to an office-scale environment.

To over come the scalability limitations, Semantic Web technologies have been combined with tuple-spaces due to the aim of bringing de-coupled, asynchronous mode of communication. Several of these approaches have been surveyed in (Nixon, Simperl, Krummenacher, & Martin-Recuerda 2008), where authors conclude that semantics-aware tuples and template matching alone are not enough to implement a semantic middleware.

In (Gómez-Goiri & López-De-Ipiña 2010), the authors propose a system based on the tuple-based message board communication mechanism extended to support Resource Description Framework (RDF) (Lassila, Swick, Wide, & Consortium 1998) triples in order to support inter-device communication. Even though the asynchronous communication model is potentially suitable, the authors conclude the proposed system faces scalability issues.

The major drawback of the semantic middlewares is that they do not allow any manipulation with the expressed knowledge, such as the process of inference resulting in reasoning with the

knowledge.

Ontology-based middlewares (Kiryakov, Simov, & Ognyanov 2002) extend the functionality of semantic middlewares by introducing the ability to reason with the knowledge about the meaning of the available information. Section 2.4.1 will provide brief description of several examples of ontology-based middlewares

With the increased scale of considered ontologies, the problem of reasoning becomes more complex. Therefore, it has been subjected to research endeavours of numerous works attempting to overcome the encountered limitations introducing concepts of distributed computing into the reasoning process. A survey of scalable reasoning techniques can be found in (Bettini, Brdiczka, Henricksen, Indulska, Nicklas, Ranganathan, & Riboni 2010). Section 2.4.2 will provide an overview of this area.

### 2.4.1 Existing Ontology-Based Middlewares

In this section we briefly introduce two examples of ontology-based middlewares.

#### 2.4.1.1 SOCAM

SOCAM (Gu, Wang, Pung, & Zhang 2004) is a context-aware middleware based on Web Ontology Language (OWL) (McGuinness, Van Harmelen, et al. 2004). OWL is used to represent the context of the environment devices operate in and reason about its changes so that the devices can adapt accordingly.

The novel feature present in SOCAM is the ability to express the quality of context. Such functionality is highly desirable in implicitly dynamic nature of pervasive computing and possible flaws in sensing technology. Expressing quality is achieved by specifically designed extensible ontology for modelling the quality of information, which assigns number of parameters for a given context. This information is then included in the reasoning process.

While SOCAM is interesting due to easily obtained extensibility, it is not clear how it would address the issues originating from massive-scale deployment. Another drawback is that it is left to the clients and devices to perform the actual reasoning.

In contrast, Axon is built with a large-scale deployment as a primary objective. Additionally,

Axon will provide the reasoning functionality to its clients. Furthermore, thanks to modular approach to dynamic knowledge, Axon maintains the ease of extensibility.

The ontology model used by SOCAM is similar to Axon as it also distinguishes between Upper Ontology and Domain-Specific Ontologies. However, SOCAM is focused on allowing communication of devices operating in multiple and frequently changing contexts typical to pervasive computing envisioned in (Satyanarayanan 2001) and more recently surveyed in (Baldauf, Dustdar, & Rosenberg 2007). Because of this focus, SOCAM models only concepts such as Location, Person and Activity, but brings no notion of more general concepts such as time or causality which are supported in the concept of Axon.

#### 2.4.1.2 Reasoning with Probabilistic First-Order Logic

Similar system for context acquisition, representation and utilisation by applications in smart spaces is presented in (Qin, Shi, & Suo 2007). The major difference from the previous example is supplementing OWL with first-order probabilistic logic (FOPL) (Nilsson 1986) into the formal model used for context description.

The motivation for combining these two approaches comes from the inability to represent uncertainty present in majority of ontology-based systems and the inability to describe semantic relationships between context entities by logic-based systems.

Thanks to FOPL, the predicates yield more values than True and False, Instead of discrete values, they return a probability of given predicate being true or false. Consequently, OWL is used to define special predicates *hasProbValue()* with two mandatory arguments: *hasContextLiteral()* representing some other predicate which probability value is examined and *hasProbValue()* representing the actual probability value. The *hasProbValue()* predicate is then used as common predicate when reasoning about context.

The context reasoning is built on rule-based inference mechanism similar to inference mechanism described in section 2.1.2. However, rules are extended by probabilities, constraints and restrictions of predicates. Constraints express assumptions about environment that are relevant for the reasoning process. Conditions allow specifying additional predicates that are considered.

Rules are subsequently extended by the definition of a dependency relationship between the particular data type and OWL object properties. This is beneficial as it allows expressing

contextual dependencies between different rules.

After evaluating a prototype implementation, the authors conclude their system is applicable only for usage scenarios that are not time-critical. The reasoning process is highly dependent on the size of the context data set. A large data set implies processing graphs with high number of nodes that have proven to be directly responsible for the delays.

Axon does not provide support for probabilistic logic and native expression of dependencies. However, the notion of dependency can be modelled by designating a particular predicate to model the dependency. This approach would not induce the need of maintaining and processing large graphs, and therefore imposes no potential bottlenecks on the overall system's performance.

### 2.4.2 Distributed Reasoning

Reasoning with large and/or multiple knowledge bases is a computationally complex task. With the advance of distributed computing, substantial research effort has been invested into investigating the possibilities of introducing distributiveness into the problem of reasoning to allow more efficient operation of reasoning systems. This section will briefly summarize the area of distributed reasoning.

#### 2.4.2.1 DRAGO

DRAGO (Serafini & Tamilin 2005) is a system aiming to build a scalable ontological reasoning tools for the Semantic Web. It implements the distributed reasoning principle by considering multiple separate ontologies. The novel approach of DRAGO lies in performing reasoning with partial ontologies separately. The results of local reasonings are then combined via semantic mappings. The whole reasoning process is a novel tableau-based reasoning procedure developed by the authors of DRAGO.

The architecture of DRAGO is based on nodes interconnected in a peer-to-peer (P2P) fashion. Each peer allows creation, modification and removal of ontologies and related mappings. In addition, each peer offers reasoning services providing access to the reasoning functionality. Ontologies are identified by URIs used to address peers containing required ontologies.

From (Serafini & Tamilin 2005), it is not clear how exactly are the ontologies distributed across the peers, as well as what is DRAGO's overall performance.

When compared to DRAGO, Axon requires more fine-grained manipulation with knowledge base based on a per-term basis. In addition, the concept of separate ontologies is not aligned with the knowledge model of Axon. Even though Axon's knowledge base is modular, it is still considered a joint, monolithic knowledge base.

#### 2.4.2.2 Reasoning with MapReduce

The possibility of using MapReduce to implement distributed reasoning in the context of Semantic Web is explored in (Urbani, Kotoulas, Oren, & Van Harmelen 2009). The reasoning is implemented using a technique for materializing a closure of a RDF graph based on MapReduce.

The closure of a RDF graph is obtained by iteratively applying RDF inference rules until no new data is derived. Application of RDF rules is encoded by a sequence of MapReduce jobs. This approach is evidently inefficient, therefore the authors provide several non-trivial implementation improvements.

However, due to the nature of MapReduce, the described system is more suitable for off-line, analytical reasoning with extra-large large ontologies, rather than more interactive mode of operation required by Axon.

#### 2.4.2.3 DORS

DORS (Fang, Zhao, Yang, & Zheng 2008) is a system attempting to introduce a practically feasible implementation of a system offering reasoning with large quantities of instance data in the context of Semantic Web.

In contrast to employing ontology mappings referred to in (Serafini & Tamilin 2005), the authors propose a distributed ontology reasoning algorithm itself. The core idea is to replicate frequently applied rules present in the ontology to each of the nodes present in the system, while letting each node reason using specific sub-set of rules. This leads to the necessity of exchanging results of the reasoning between separate executions of the reasoning algorithms.

The evaluation of a prototyped DORS system has shown the proposed is able to handle large ontologies better than previously proposed systems. However, the authors conclude DORS is not well suited for coping with ontology updates.

In contrast, a typical use-case scenario of Axon will imply frequent changes of the underlying ontology (e.g. new sensor readings), therefore Axon is able to handle frequently-updated ontologies.

#### 2.4.2.4 P2P Reasoning

The P2P reasoning system presented in (Anadiotis, Kotoulas, & Siebes 2007) also considers the environment of Semantic Web and distributes ontologies using a DHT. However, the novel approach lies in aiming for more coarse granularity. Rather than splitting the ontology into triples, the authors propose to split the overall ontology into multiple smaller ontologies and let every peer participating in a P2P overlay (determined by DHT) retain control of the ontologies it is responsible for. This way, the authors attempt to achieve better performance of the reasoning process.

Better reasoning performance is obtained by splitting an original query into sub-queries and letting different peers handle each sub-query. Upon combining the sub-results, the system can decide if the answer is sufficient and optionally reformulate the queries in order to obtain better answers.

Based on the specific character of used data-set, the authors conclude the increased capacity to perform complex local reasoning is not fully utilised. On the other hand, the ability to retain control over ontologies is considered useful.

Axon is similar to the P2P reasoning system in a sense that Axon also splits and distributes present ontologies according to a DHT, but the partial ontologies are still considered fragments of globally monolithic ontology. Additionally, the reasoning model of Axon is iterative and it is not based on partial query resolution.

## 2.5 Concluding Remarks

The relevant state-of-the-art for this work can be summarized into two categories. The first one contains ontology-based middlewares aiming on device interoperability but neglecting the aspects of performance and scalability. The second one addresses the issues of scalable and distributed reasoning via bulk processing of large knowledge bases with little emphasis on

| | Knowledge Representation | Reasoning | Key Concept |
|---|---|---|---|
| Ontology-based MWs | | | |
| SOCAM | OWL | By applications | Knowledge facilitates adaptation to context. |
| FOPL | OWL | Probabilistic logic | Probabilities help dealing with uncertainties. |
| Distributed Reasoning | | | |
| DRAGO | Description Logic | Tableau-based | Reasoning with partial ontologies. |
| MapReduce Reasoning | RDF | RDF rules | Inference as sequence of MapReduce jobs. |
| DORS | OWL | Description Logic | Local computation and propagation of results. |
| P2P Reasoning | RDF | RDF rules | Peer cooperate to resolve queries. |

| | Advantages | Disadvantages |
|---|---|---|
| Ontology-based MWs | | |
| SOCAM | Extensibility | No support for general concepts (e.g. causality) |
| FOPL | Rule dependency modelling | Inefficient for large data sets |
| Distributed Reasoning | | |
| DRAGO | Well-paralelisible computation model | Complex procedure of materialising the results |
| MapReduce Reasoning | Adopted MapReduce Scalability | Lack of interactive operation |
| DORS | Efficient work sharing | Necessity to transfer a lot of data |
| P2P Reasoning | DHT-based architecture | Most of the queries are locally resolvable |

Tabela 2.5: Comparisson of investigated ontology-based middlewares and distributed reasoning.

interactive operation. The final comparisons of all investigated solutions can be found in Table 2.5.

In this thesis we will be focused on designing a system that will be positioned between these two categories. The system we propose will belong to the category of semantic middlewares because it will allow describing the meaning of the data it contains by expressing it in the form of Prolog facts and rules. At the same time, the proposed system will be classified as an ontology-based middleware because the Prolog inference will allow reasoning with the data contained in the system.

The differentiating aspect of the proposed system will be its distributed character, thanks to which we will try to remedy drawbacks of both semantic and ontology-based middlewares. Our system will be designed to offer the reasoning functionality to massive amounts of clients in an interactive manner while maintaining the capacity to handle large volumes of data.

## Summary

In this chapter we have provided an explanation of what is understood under the terms *knowledge* and *reasoning*, how these concepts related to distributed computer systems and logical programming in particular. Consequently, we included a brief overview of DHTs. Then we have provided a description of a conceptual system providing a foundation for the implementation of a prototype presented in this thesis. Finally, we have explained the position of this thesis with the recent state-of-the-art and identified its differentiating features from other existing solutions.

In the next section, we will describe the design of a system attempting to answer the research question being answered by this thesis.

# 3 System Architecture

In this chapter we describe the solution of the problem addressed by this thesis. The chapter first provides a general overview of the proposed architecture in section 3.1. The organisation of the remainder of this section follows the decisions made during the design process. Each step is explained in detail, including the statement of requirements relevant for the particular step being described. We chose this approach in order to clearly explain the motivation and reasoning behind each decision made. In particular, Section 3.2 explains how we tackled the problem of distributing a Prolog knowledge base. Section 3.3 describes the computational model of the proposed system. Finally, Section 3.4 discusses particular aspects related to distributed architecture of the system.

## 3.1 General Overview

The foundation of the proposed system is constituted by a Prolog engine. As we showed in Section 2.1, Prolog is a suitable tool for implementing the concepts of *knowledge* and *reasoning*. In our scenario Prolog terms will be used to describe knowledge about the known world. Terms will appear in form of facts representing simple statements, or as rules representing relations between facts or describing theories presented in 2.2. The database of Prolog terms will therefore consitute a knowledge base and Prolog's inference engine will provide the reasoning with the knowledge. The actual machine-to-machine communication will be performed by asserting terms into the knowledge base and issuing queries to the Prolog engine.

The first design idea lies in selecting an attribute of any Prolog term that can be used as an input for the DHT's hashing function in order to achieve uniform and therefore efficient distribution of terms across the nodes in the system. With this approach, it will be possible to find the location of any given term within the system with constant algorithmic complexity and consequently ask the identified node to perform the given operation regarding the term.

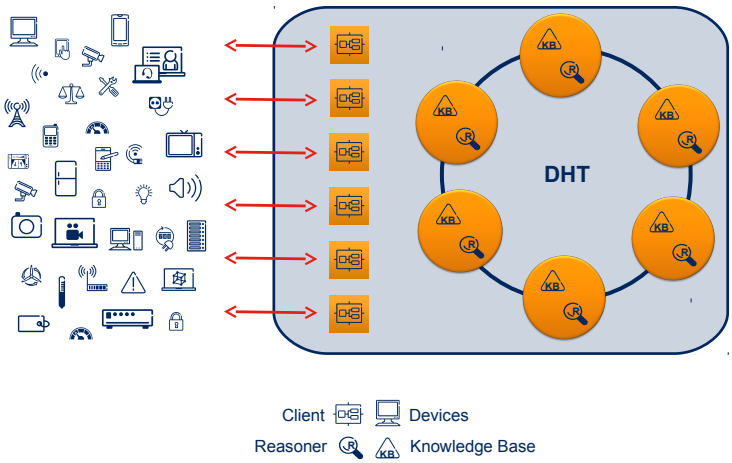



Figura 3.1: General system overview.

The motivation for this decision is to limit the data movement which can potentially produce significant overhead. Instead, comparably smaller context capturing the state of Prolog goal solving will be migrating among the nodes, thus effectively bringing computation to the data.

The general system architecture can be seen in Figure 3.1. The middleware consists of multiple nodes interconnected in a ring structure similar to Chord (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001), Dynamo (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007) or Cassandra (Lakshman & Malik 2010). Each node is composed of two main components: a storage component and a reasoner component.

- The *Storage* component is responsible for managing the persistent storage of the terms belonging to node's portion of the DHT key-space range.
- The *Reasoner* is responsible for answering any incoming requests. This can involve requesting a storage operation from the storage module, executing actual computation related to the solution of the queried Prolog term, or sending the computation to another node if needed.

The client nodes can issue operation towards any of the middleware nodes, or they preemptively ask for the current middleware topology and issue requests to appropriate nodes

directly.

## 3.2 Sharding a Prolog Knowledge Base

This sections explains the concept of splitting a monolithic Prolog knowledge base into parts and consequently distributing them across nodes present in the system.

### 3.2.1 Storage-Specific Requirements

In one sentence, the storage back-end must be able to effectively store large number of relatively small chunks of data (e.g. Prolog terms). This generic requirement can be further specified by the following properties.

*Uniform Load Distribution.* Distributed systems aiming for good scalability must ensure that all nodes participating in the system are uniformly utilised. Applied to the context of this chapter, this can be rephrased as:

**Requirement 1.** *In a system where $N$ nodes are responsible for storing a total of $T$ prolog terms, each node holds approximately $T/N$ terms.*

Ensuring this property prevents the creation of hot-spots in the system, meaning that there are no over-utilised nodes. In addition, maintaining this property allows achieving certain degree of flexibility in the system: it is possible to remove nodes from the system without facing the risk of disrupting its operation by removing an important node, as well as easily add new nodes without hitting any limitation imposed by the system architecture.

*Explicit Addressing.* The data placement mechanism should be exposed and usable by the rest of the system. There are scenarios where an application can benefit from having the information about data placement algorithm, for example by delegating any computation using the data to a node actually storing the data. In the context of the proposed system, this requirement means that given a Prolog term, it should be possible to determine which node is responsible for storing it.

*Reasonable Granularity.* Prolog terms are relatively small chunks of data. Therefore considering a Prolog term as an elementary unit of data might not prove to be the most efficient solution. The need for grouping individual Prolog terms into larger chunks is therefore apparent.

This grouping is different from the one described in Section 2.2 because Axon demands modularity on a higher level of abstraction (modules of knowledge are oblivious to the actual implementation). The storage system works on a lower level. It considers the implementation and distinguishes modules based on syntactical or structural properties of stored terms.

*Efficient Lookup.* This requirement is closely related to the previous one. When determining a location of some data, the provided answer should be unambiguous. More specifically:

**Requirement 2.** *Given a Prolog term $T$, performing operation $owner(T)$ must return exactly one answer.*

The origin of this requirement lies in Section 2.2. Any knowledge represented by the terms should be modular, therefore it is desirable to involve as few nodes as possible in the computation related to the manipulations with terms on the modular level. Moreover, unambiguous addressing removes the burden of complex routing logic being implemented in the application and allows a simpler and cleaner solution.

Unique assignment of terms to nodes from the application point of view does not prevent the storage solution from introducing replication mechanisms and therefore disobeying Requirement 2, as long as this remains hidden from the application.

Finally, the actual implementation of the $owner(T)$ should be inherently efficient. It should be implemented with the complexity of $O(1)$ and should not introduce unnecessary overhead or single point of failure, e.g. network communication such as directory lookup.

*Persistence and Fault Tolerance.* The Prolog terms represent knowledge. As such, the knowledge is persistent. It is not acceptable for the knowledge to disappear from the system in case of a failure. The terms should be stored in a persistent way, thus providing resiliency against temporary crashes and breakdowns of a node responsible for them. Furthermore, in scenarios when a node fails completely, the terms should be somehow recoverable, for example from their replicas stored in other nodes in the system.

*Easy access.* The proposed system is expected to allow high throughput operation. For this reason, the storage solution should not introduce unwanted overhead on accessing the data. At the same time, the interface provided to the rest of the application should be kept as simple as possible, in order to help keeping the application logic minimal and efficient.

### 3.2.2 The Choice of Distributed Hash Tables

Having examined several other frameworks, such as MapReduce (Dean & Ghemawat 2008) and Pregel (Malewicz, Austern, Bik, Dehnert, Horn, Leiser, & Czajkowski 2010), relevant for processing massive amounts of data led to conclusion that none of them is suitable for building an interactive system. Existing solutions described in Section 2.4.2 are optimised for batch rather than interactive mode of operation, i.e. they emphasise the throughput in scenarios where we low latency is also desirable.

Therefore we chose distributed hash tables as the underlying concept to build our storage back-end.

As explained in Section 2.3, hash tables work with the key-value pairs. Therefore the first step in explaining this choice is describing the proposed mapping of Prolog data model onto the key-value concept.

It has been shown previously in Section 2.1.4 how every Prolog term is described by its name and arity. One of the core ideas of this work is to propose the following mapping:

**Proposal 1.** *Every Prolog term $T$ named $t$ and of arity $a$ can be stored as a value in a hash table addressed by a key composed by concatenating $t$ and $a$.*

For example, given three Prolog terms

```
termA(atom1).
termA(atom1, atom2).
termA(atom2, atom3) :- termA(atom1, atom2).
```

performing operation $get(termA.2)$ would return only the terms named $termA$ and of arity equal to 2:

```
termA(atom1, atom2).
termA(atom2, atom3) :- termA(atom1, atom2).
```

This example, however, exhibits a collision of multiple values and therefore not completely fulfilling Requirement 2. Typically, hash tables and applications built atop them strive for avoiding collisions wherever possible.

Collisions are, naturally, unwanted phenomenon also for this work. To remedy this situation, the mapping in Proposal 1 has to be generalised:

**Proposal 2.** *Every group of Prolog terms $T_1, T_2, \cdots, T_N$ of name $t$ and of arity $a$ can be stored as a value in a hash table, addressed by a key composed by concatenating $t$ and $a$.*

This generalisation will remove any collisions introduced by Proposal 1. Moreover, it will contribute to more efficient preservation of Prolog semantics, as follows.

*Preserving Prolog semantics.* The division of terms according to their names and arities is native to Prolog. Any time some computation step is performed, it considers only currently known predicates of given name/arity. Therefore, grouping multiple values (i.e. terms) under one key is not problematic but, on the contrary, opens a possibility for the Prolog inference algorithm to effectively access all terms required for the given step.

Another important semantic aspect of Prolog is the ordering of terms described in Section 2.1.4. Because of Proposal 2, all the related terms are already grouped. However, it is left to the application logic to ensure the correct ordering of terms under a given key. There might be operations conditioned by runtime context (e.g. deleting terms with variables in arguments) that require more than just the insertion or removal of first or last term of the given group. It is therefore impossible for the storage system to provide this functionality.

### 3.2.3 Fulfilling the Storage-Specific Requirements

It was shown how it is possible to model Prolog data model using DHTs. Now it is necessary to assess how can the proposed mapping answer the formulated storage-specific requirements.

*Uniform Load Distribution.* Provided the hashing function of a chosen DHT implementation is balanced, the DHT will natively provide balanced load distribution. The remaining threat to disrupting the uniformity can come from actual instances of the stored terms. It might happen that the terms of certain names and arities will be more frequent than others, thus creating hot-spots on nodes responsible for storing them. However, it is assumed that the scale and total amount of terms present in the system will outweigh any imbalance in reall-world use-case scenarios.

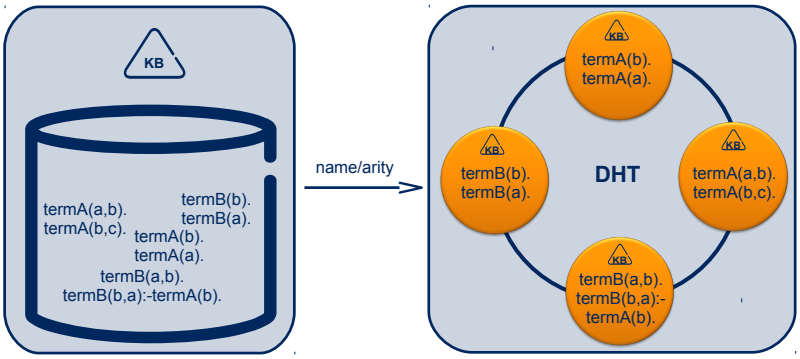



Figura 3.2: Illustration of the proposed sharding approach.

*Explicit Addressing.* Name/arity are easily extractable from any prolog term by the application. Using this information to address a node responsible for the data is therefore straightforward. Alternatively, this mechanism could also be encapsulated in data access libraries.

*Reasonable Granularity.* The mapping proposed in Proposal 2 ensures the terms will not be treated separately but rather grouped according to Prolog semantics.

*Efficient Lookup.* The choice of a key for the key-value pair guarantees there will be exactly one node responsible for the given key. In addition, hashing function guarantees the desired lookup complexity.

*Persistence and Fault Tolerance.* Modern DHT implementations usually provide both persistent storage and fault tolerance. For example, instantiating a Cassandra cluster with a replication factor extends the out-of-the box persistent storage by replicating the data on neighbouring nodes.

*Easy Access.* Due to their distributed character, DHT implementations also usually provide high throughput, as well as simple API.

### 3.2.4 Recapitulation of Proposed Sharding Approach

This chapter described how it is possible to thoughtfully split a potentially very large set of Prolog terms into a more structured topology that can provide effective and scalable storage back-end for a distributed system. The ideas of this chapter are illustrated in Figure 3.2.

## 3.3 Migrating Computation

The storage solution described in the previous section induces implications necessary to be considered when designing the consequent computational model. This section describes what the proposed computational model is and how it is integrated in the proposed data model.

### 3.3.1 Computation-Specific Requirements

The computational model should have the following properties.

*Limited Data Movement.* It is assumed the system will have to operate with large amounts of data. In addition, a single user request (e.g. solving a Prolog goal) can potentially require access to large amount of data (e.g. many terms related to the solution process). This can potentially lead to several problems:

- Moving data can be slow. The data would have to be sent over a network to a different node. While this behaviour could be tolerable provided the underlying infrastructure is of high performance, it is not safe to consider such environment for granted.
- Redundant data replication. In some scenarios, data redundancy can lead to increasing overall performance of the system. This, however, applies only in specific scenarios, and particularly in real-domain workloads. Therefore it should be carefully examined whether this approach is viable solution for the problem being answered by this thesis.
- Excessive memory operation. The expected volume of data present in the system is far greater than what can be placed in the operating memory of a contemporary computer system. Relying on all the data necessary for the computation being available locally can therefore not be achievable.

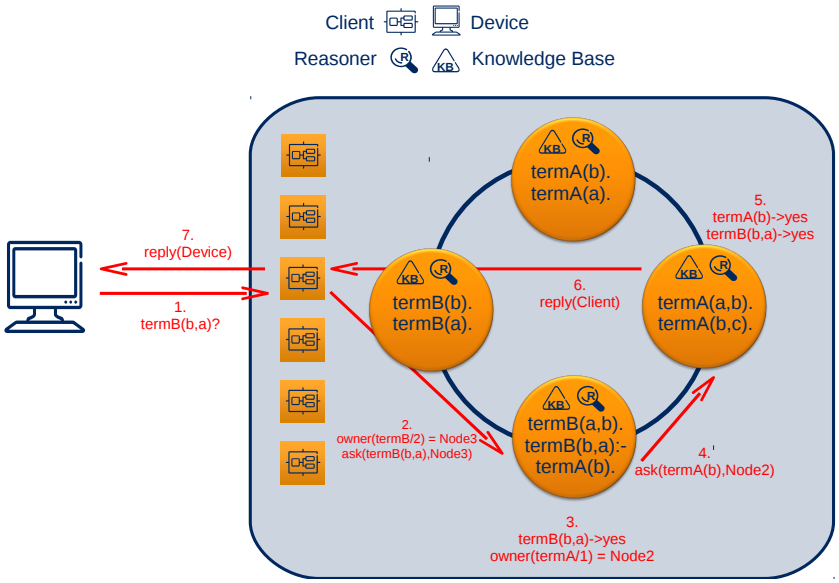



Figura 3.3: Prolog goal solving extended with the concept of migrating computation.

- Consistency issues. The proposed data model implies a single node being exclusively responsible for handling read and write requests regarding terms of given name and arity. Introducing replication of this data at different nodes would require explicit management of the replicated data.

It is therefore desirable to strive for limiting the amount of data that needs to be manipulated.

*Facilitate Reactive Behaviour.* Particular usage scenarios, such the one described in Section 2.2, could benefit from reactive mode of operation with low latency. This means that when some event occurs (e.g. an assert operation renders some Prolog goal solvable), it should be possible to notify the application.

With the aim of fulfilling most of the properties mentioned above, this work proposes a computational model based on migrating the computation among the back-end nodes of the system. The overall operation of the system is illustrated in Figure 3.3.

A Prolog computation will therefore go as follows:

1. A device issues Prolog query asking if goal $termB(b, a)$ is true.

2. A client receives the requests and examines the query. It reads the name and arity of the

term being queried ($termB/2$) and determines a back-end node responsible for the term (*Node3*). The client then forwards the query to the corresponding node.

3. The reasoner at *Node3* receives the query and instantiates a new Prolog engine to handle the query. If the library containing terms $termB/2$ is not loaded, the reasoner at *Node3* loads the the library with the assistance of the storage daemon and then initiates the goal solving and finds out $termB(b, a)$ is present in the knowledge base. Then it continues with the next goal, $termA(b)$. However, *Node3* is not the owner of terms $termA/1$.

4. The computation has to be sent to *Node2* which is responsible for terms $termA/1$.

5. *Node2* receives the computation, instantiates a new Prolog engine and loads it with the context related to previous computation. It successfully solves goal $termA(b)$. and discovers this concludes the whole solution process based on the information present in the computation context received from *Node3*.

6. *Node2* can therefore provide a final answer to the client.

7. Finally, the client can forward the received answer to the device.

The proposed computational model introduces an apparent drawback. The process of determining ownership of terms, as well as sending the context over the network inevitably causes additional overhead. However, with respect to the thesis stated at the beginning of this work in Section 1.1, the assumption is that this overhead can be outweighed by the obtained advantages. The advantages of the proposed computational model are following:

- Splitting the computation needed to answer one query allows sharing the resources of one node among multiple concurrent queries. This can pay off provided the number of concurrent queries is sufficiently large.

- Partitioning the computation into smaller steps provides an opportunity for check-pointing the solution, what makes it possible to introduce fault-tolerance into the system.

- The computation context is far smaller than the data the computation works with. Transferring smaller amount of data is naturally more efficient. More details about exact information constituting the computation context can be found in the relevant implementation Section 4.2.

- Keeping the data at one node without any explicit replicas facilitates data consistency.
- The static data is not redundantly stored.
- The proposed approach is relevant for read requests only. Write operations are forwarded by Clients directly to nodes responsible for terms being inserted or removed.
- Each node can load into memory only the data immediately needed for the computation. This prevents any problems associated with the memory limitations.

### 3.3.2 Recapitulation of the Proposed Computation Migration Concept

This chapter has shown how the proposed data model can be supplemented by a computational model in order to achieve the desired functionality. The core idea is to instantiate a Prolog engine on each of the nodes present in the system and let each instance execute part of the computation related to the data already present at the given node, thus achieving the concept of decentralised and collaborative computation of Prolog Clauses.

## 3.4 Facing the CAP Theorem

Still, the system described so far suffers from one important drawback. Figure 3.4 illustrates the problematic scenario.

In the scenario, *DeviceA* issues a request that initiates a goal solving at *Reasoner4* (step 1). The solution requires to be forwarded to reasoners *Reasoner2* and *Reasoner1* during steps 2 and 3 respectively. Solution at *Reasoner1* then takes longer time, sufficient for another request from *DeviceB* to reach *Reasoner2* in step 4. This request can potentially change terms related to the solution of the first request. Therefore, by the time the answer to the first query is returned to *ClientA* during steps 5 and 6, the provided answer might no longer be correct.

This situation is a manifestation of the CAP theorem (Gilbert & Lynch 2002). Achieving consistent results of goal solving is not possible while maintaining the whole system available and partition-tolerant.

This section describes how is the proposed system preventing this situation to happen.

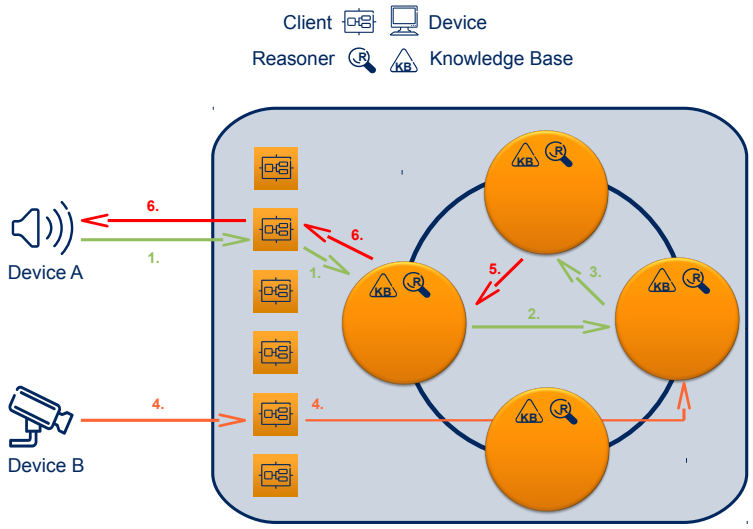



Figura 3.4: Example of inconsistent goal solution.

### 3.4.1 Assessing Consistency, Availability and Partition-Tolerance

The first step towards solving the problem with inconsistent queries was to revisit the envisioned deployment scenario described in 2.2 and assess what is the relation between Axon and CAP theorem. It was concluded that:

- *Partition-tolerance* has the highest priority. One installation of Axon assumes one consistent and monolithic knowledge base. It is therefore mandatory to ensure that every query has the up-to-date view of the whole knowledge base.

- *Consistency* has the second highest priority. It was concluded that partial or best-effort answers are not desirable. Once the system provides answer, it must be guaranteed that the answer is correct relative to the point when the query was issued.

- *Availability* has been designated as the attribute with the lowest priority. It is acceptable for the clients to wait for a reasonable amount of time with no reply from the system.

### 3.4.2 Towards Consistent Queries

The data model described in Section 3.2 provides solid foundation for fulfilling the assessed priority of partition-tolerance, consistency and availability. It can implicitly ensure consistent data for all nodes provided that read and write requests are managed adequately. This can be achieved in multiple ways.

The first approach can be described as invalidation propagation. With this approach, the migrating query resolution would leave behind some footprint, so that it would be possible to track all queries that used particular term and notify them once the term gets changed. The main benefit of this approach is the optimistic query resolution. Unless the terms used in a query actually change, no additional computation is required, what contributes to faster operation of the overall system. However, in case of frequent term updates, combined with long-running queries, the frequent invalidation would produce exceeding amount of messaging caused by the broadcasted invalidate messages, as well as potentially prevent some queries from being answered completely. Even though availability is the least prioritised attribute, this behaviour is still unacceptable.

An alternative approach is to introduce certain degree of global synchronisation into the system. In this approach, the system operation would be split into two phases: a *read phase*, and a *write phase*, what is a division similar to the tuple-based communication described in Section 2.4. Each phase would allow only one kind of operations to be executed. As a result, there would be a guarantee that while a read phase is ongoing, the underlying data will not be changed by any incoming write request. More detailed explanation regarding the implementation of phase based operation can be found in Section 4.3.

The phase-based operation is rougly equivalent to spanpshot isolation with versioning using and clocks,

### 3.4.3 Recapitulation of the Position Against the CAP Theorem

To summarise the position of the proposed system towards the CAP theorem, the proposed system can be classified as CP because it prioritises consistent data model with no partition tolerance re-check of partitions at the cost of limited availability because it requires consistent an monolitihic knowledge base and consistent queries at the cost of limited availability.

## Summary

In this chapter we have described the architecture of the proposed system. For every decision made we provided detailed description of what were the specific requirements we attempted to answer with the given decision and how these requirements met. We chose this way of explanation in attempt to make our decisions more comprehensive.

The proposed distributed system organises nodes in a ring-like structure built atop a DHT, thus allowing efficient balancing of stored data and performed computation. System operation is split into two phases in order to ensure consistency of the system at the cost of availability.

In the following section, we will describe several key aspects related to the implementation of the proposed system.

# 4 Implementation

In this section we provide more detail on implementation of particular aspects of the system proposed in Chapter 3. In addition, we mention several early improvements resulting from the chosen implementations and early observations of the system operation.

The chosen implementation of underlying DHT can be found in Section 4.1. The implementation details of the migrating computation concept can be found in Section 4.2. Implementation of phase-based operation is more closely described in Section 4.3. Finally, Section 4.4 presents several early-made optimisations regarding the implementation of the proposed system.

## 4.1 Chosen DHT Implementation

As it was stated in Section 2.4, the purpose of this work is to provide a prototype for a complex system, not to design an application specific storage solution from scratch. Therefore an existing implementation of DHT was chosen to meet the described requirements.

The solution of choice in this work is Apache Cassandra (Lakshman & Malik 2010). It is a well-established and well-supported project that brings all the properties desired by the proposed system.

Namely, it exposes the partitioning functionality it uses for data placement, so that applications can utilise this information. Moreover, it efficiently handles persistence and fault tolerance. Therefore it is possible to consider the key proposed in Proposal 2 as equivalent to Cassandra's row key.

In addition, Cassandra's feature of ordering entries by secondary keys can be utilised to efficiently implement ordering of terms under given key. In order to achieve this, the column family for storing terms should look like this:

```
String  name/arity primary key
```

```
Long    timestamp secondary key
String  terms
```

With this data model, the write(i.e. assert), delete(i.e. retract) and read(i.e. query) operations can happen according to the following algorithms.

**Input** : *name/arity* : String
*append* : Boolean
*timestamp* : Long
*term* : String
**Output**: *void*

*Procedure:*

**if** *not append* **then**
  *timestamp* = (−1) * *timestamp*;
**end**
**put**([*name/arirty*,*timestamp*], *terms*);

**Algorithm 1**: Term assertion.

The input terms are expected to be ordered correctly by the application with the respect to the desired append value. Appending terms is implemented by inserting a *timestamp* greater than any existing ones. Prepending is implemented by negating the input *timestamp* resulting in a number smaller than any existing ones. Ordering on secondary key then ensures the correct ordering of terms.

**Input** : *name/arity* : String
**Output**: *terms* : Term[]

*Procedure:*

*terms* = **get**([*name/arity*],_);
**foreach** *term t in partialTerms* **do** *terms* ∪ = *t* ;
**return** *terms*;

**Algorithm 2**: Retrieving terms from the knowledge base as a list.

When queried by primary key only ( character '_' represents arbitrary secondary key), Cassandra returns all known entries for the given key. These entries are correctly ordered thanks to being properly inserted by algorithm 1 therefore it sufficient to straightforwardly concatenate them into one block of terms and return this block to the application. The implementation of the *retractTerm()* method is done by the underlying Prolog engine, which processes the set of therms and actually removes the appropriate term.

The storage back-end is not able to perform term retraction autonomously. Therefore the

**Input** : *term* - term to be retracted
**Output**: *none*

*Procedure:*

*allTerms* = **read**( *term*.getNameArity() )
*newTerms* = *allTerms*.retractTerm(*term*)
**write**( *newTerms*.getnameArity(), *System.getTime()*, *newTerms*)

**Algorithm 3**: Term retraction

retraction consists of three steps. The application has to load all know terms first, then retract the appropriate term, and finally write the modified set of terms.

## 4.2 Migrating Computation Context

We used the tuProlog (Piancastelli, Benini, Omicini, & Ricci 2008) as the basis for our reasoning engine. We have extended tuProlog with the functionality of checking the solved terms being locally resolvable and conditionally migrating the solution context to the node responsible for the next goal. However, built-in predicates such as list constructors or term *true/0* have been replicated in all reasoners in order to prevent unnecessary migration.

In order to implement the computation transfer, the computation context packaged in the messages exchanged between different *Reasoners* must contain:

- Any contextual information associated with the Prolog semantics. This includes Prolog variables (their names and bindings) and sub-goals needed to be solved in order to obtain a solution for the top-level goal, with the order in which they need to be solved included. The reasoning modules performing parts of the computation will naturally modify this context. For instance, they might extend the sub-goals by new sub-goals determined from local terms, as well as significantly re-order the whole sub-goal set (e.g. due to encountering the cut operator).

- Identification of the *Client* node awaiting the final solution.

- Trace of the solved goals. This feature is not natively provided by Prolog and has to be explicitly supplemented to the context.

## 4.3 Phase-Based Operation

In order to satisfy the described requirements regarding the CAP theorem, the system proposed in this work will implement the alternative suggesting phased mode of operation. This decision brings several implications that require close examination and explanation.

*Buffering requests.* Execution of each of the phases will inevitably take certain amount of time. During this time, it is possible that new client requests will arrive. The incoming requests can not be directly served because continuous arrival of requests matching the current phase would cause the phase to never terminate. For this reason, the system should allow buffering the incoming requests, thus delaying their handling until the next instance of relevant phase begins.

In addition, as implied by Prolog semantics about ordering of terms, the buffers must preserve the correct ordering of incoming write requests for each group of terms about to be modified.

*Leader Election.* The introduction of phase-based operation required node synchronisation. For this purpose we implemented a leader election algorithm based on the Paxos algorithm (Lamport 2001) using the Zookeeper (Hunt, Konar, Junqueira, & Reed 2010) coordination service. The synchronisation is achieved by nodes reporting end of each phase. The leader node awaits for reports of all (alive) nodes and triggers a phase change once all nodes have finished their current phase.

While the leader election procedure is not strictly required in order to implement barrier-based synchronisation, the role of a leader is potentially useful for the implementation of additional features, e.g. fault tolerance of query resolution.

## 4.4 Early Improvements

Already during the first works on implementing the proposed system, several previously unforeseen problems became apparent. In order to remedy them, multiple optimisations had

to made even before the proper measurement and evaluation phase could begin. Designing and implementing these improvements constitutes a substantial amount of work done on this project. This section will briefly describe the reasons, solutions and impact of some of the most relevant improvements done to the system.

### 4.4.1 Asynchronous Messages

Originally the proposed system used exclusively synchronous messages. However, the necessity to wait for an answer has the potential to drastically impair the overall throughput of the system. For example, with the phase-based mode of operation introduced in section 3.4, the front-end node would be allowed to issue at most one operation per phase per back-end node. This is unacceptable, particularly in case a query phase takes longer time than average.

To answer this issue, the possibility to issue asynchronous requests has been introduced. It is now possible for the front-end node to decide which mode is necessary to handle client's request. Typically, asynchronous messages are relevant for write operations due to high throughput provided. Read operations, on the other hand, are most suitably implemented using synchronous messages because consistent and correct result is more desirable than partial best effort answer.

The most significant impact of introducing asynchronous messaging to the rest of the system lies in the increased complexity of incoming operation buffers presented in Section 4.3.

### 4.4.2 Managing Request Buffers

The increased rate of impending requests caused by allowing asynchronous messages led to revisiting how the requests are buffered and processed. The time required to process all elements in the buffers is proportional to the number of buffers present. Therefore, increased number of requests led to prolong the query relevant phase execution time, thus allowing even more requests being buffered for the next execution of the given phase. This led to an avalanche effect and inevitable stalling of the entire system.

The solution for this problem was the decision to process fixed maximal number of requests during each phase, thus effectively transforming the buffer into a FIFO queue. The benefits provided by this approach are double:

- Fixed amount of processed requests guarantees relatively short phase execution time and therefore fair alternation of of read and write phases.
- Fluent alternation of phases prevents failures possibly resulting from time-outs originating in the other parts of the system (e.g. Zookeeper session time-out)

### 4.4.3 Aggregating Asserts

Processing a batch of incoming write requests opens one more possibility for optimisation. Rather than writing each incoming term by a separate Cassandra operation, it is beneficial to group terms of the same name/arity into one insert operation and consequently group multiple insert operations into one batch.

It is necessary for the aggregation logic to preserve the ordering incoming requests due to Prolog semantics. This brings additional computational cost, but that cost is compensated by performance gains achieved from Cassandra batch operations. In addition, the batch operations are atomic, thus providing means of further improvements regarding the robustness and resiliency of the whole system.

### 4.4.4 (Un)Loading Prolog Libraries

There is a certain limit to the amount of Prolog terms that can be placed in the memory of a running system. Assuming the total amount of terms known by the system can be far larger, it is important to provide a mechanism allowing manipulation with excessive amount of data.

Grouping the terms by their name and arity offers a straightforward solution. Each group of terms can be perceived as a Prolog library [1] and swapped in and out of a running Prolog engine as necessary. The swapping process might be happening quite often, as well as it might operate on potentially big libraries. For such reasons, the need to design an effective solution is apparent.

Early observations showed that attempts of manipulating with the libraries using chunks is not a feasible approach, mostly due to inefficient re-loading of a library chunk into an already active library.

[1] Prolog library is a group of terms representing a module of knowledge present in the system.

The libraries are therefore swapped in and out of the prolog engine atomically. This brings loading times superior to chunked approach, but helps with maintaining the data consistent. A downside is that big libraries have to be stored using multiple write operations because they exceed the size allowance of single insert operations.

### 4.4.5 Limiting Computation Context State

As described in Section 3.3, one of the core principles of the proposed solution is the migration of the computational state. Because the need to send the computation to another node will arise very frequently, the whole process should be as efficient as possible.

The first optimisation possible in this case is limiting the amount of data that needs to be transferred. There is no need for the entire Prolog engine to be transferred. Instead, only the data necessary for reconstructing the computational context (such as variable bindings, next goals to solve, etc.) need to be sent away.

In addition, upon resuming the computation, there is no need to instantiate a new Prolog engine. Instead, an existing instance can be either duplicated or taken from a pool of available engines and the received state can be loaded into the engine.

Finally, third-party serialization libraries [2] can be used as a substitute for the native serialization mechanism because they offer superior performance.

### 4.4.6 Snapshotting System Topology

Both back-end and client nodes need to be aware of the current system topology in order to address requests. Originally this functionality was obtained by actively querying Cassandra processes for up-to date information.

This is apparently an overkill, since the node churn in the system is not assumed to be very high. It is therefore sufficient to regularly take a snapshot of the system topology and refer to it during normal operation. The snapshot will be refreshed in reasonable intervals. There is also a possibility to broadcast a snapshot invalidation command by a node that detects a change in

[2] protostuff - java serialization library, http://code.google.com/p/protostuff/ (accessed at June 13, 2013).

the topology the quickest from the nodes in the cluster and therefore should notify other nodes about the detected event (e.g. node failure).

## Summary

In this chapter we described the implementation of the proposed system.

First, we have explained the selection of Cassandra as the DHT implementation-of-choice thanks to its out-of-the-shelf fault-tolerant persistent storage and good scalability. Then we explained how we implemented the migration of Prolog's computational context based on determining the location of the next goal to be evaluated by using the underlying DHT. Next, we showed how we implemented node synchronisation based on phase change announcements originating from a node elected to act as a leader. Finally, we presented several early-made improvements.

The following chapter will contain the evaluation of the presented implementation.

# 5 Evaluation

This chapter presents the measurements performed while experimenting with a prototype of the proposed system. As first, Section 5.1 will describe the methodology used to evaluate the prototyped system and Section 5.2 will describe the experimental setup we used. Next, Section 5.3 will focus on the impact of overhead caused by distributed mode of operation. Following sections will present other attributes describing performance of a distributed system. Section 5.4 will deal with scalability and section 5.5 will describe load distribution.

## 5.1 Methodology

The correct way of evaluating a distributed system is to compare it to a comparable centralised system rather than 1-node installation of the distributed system itself. The centralised system used as a reference for the evaluation in this chapter was implemented based on the proposed distributed system, but stripped from any functionality and overhead originating from the aspects of distributed mode of operation.

In the remainder of the evaluation section of this work, whenever a reference to a one-node installation of the proposed system is made, it is meant to refer to the reference system described herein.

The differences of the centralised system include:

- Persistent storage is implemented using regular files (in contrast to using Cassandra).
- There is no need for using Zookeeper for phase synchronisation and leader election.
- The Prolog goal solving is not checking for goals being locally resolvable and does not implement the migrating computation.

Besides the aforementioned differences, it is essential to maintain other aspects of operation equivalent for both centralised and distributed alternatives. The most important features to maintain are:

- Approach to handling incoming requests. Accepting and processing client requests is implemented equivalently in both alternatives.
- Prolog goal solving. Besides the notion of locally resolvable goals, the remaining operation of the Prolog engine is the same and yields the same results.
- Alternating phase-based operation. Even without the concept of migrating computation, the one-node system must maintain this functionality to be considered comparable. Abandoning the phase-based operation would essentially give incomparable system, as well as compromise the guarantee of consistent queries and fair request handling described in Section 3.4.

## 5.2 Experiential Setup

The experiments attempting to evaluate this work were performed on a cluster of five machines, each offering 4-core Intel Core2 Q9400 CPU operating at 2.66 GHz and 8GB of RAM. The one-node installation of the system was deployed on one of the cluster machines as well. The workload was achieved by deploying client instances on the same nodes, thus leveraging fast local network connectivity of the cluster. Because the backend process was utilising only one processor core, we were able to instantiate up to 3 client processes per physical node while minimising the introduced interference between the reasoner and the clients.

The preliminary experiments with the described reference system have shown that, given the available hardware setup, the one node installation of the proposed system can hold up 400,000 Prolog terms while being able to serve 4,000 requests per second. The absolute values by themselves provide little information, but are an adequate starting point for evaluating the performance of the distributed system.

We used two data sets during our experiments, both of which were generated to simulate a real-life knowledge base. The first one consisted of 400,000 systematically generated Prolog terms, including averagely 5-hop dependencies among each other, what is the upper limit of

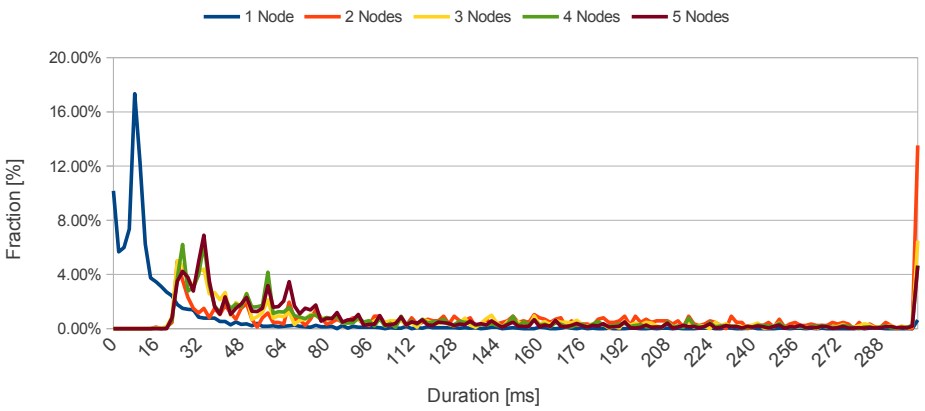



Figura 5.1: Distribution of read phase durations.

terms sustainable by the one-node system given the available hardware setup. The second data set consisted of 1,500,000 terms. The terms consisted of both facts and full clauses. The clause bodies contained other atoms and clauses present in the knowledge base, creating references of averagely 5 levels. We observed the system behaviour during read and write phases accessing random terms from the data set.

## 5.3 Observed Overhead

During this set of experiments, we have measured how the one-node system operates until the workload exceeds its limits (thus acquiring the information about maximal capacity stated in Section 5.2). Then we examined how different installations of the distributed system operate given the same workload. The analysis is split into two parts: the first one for read phases, the second one for write phases.

### 5.3.1 Read Phase Analysis

Figure 5.1 shows the distribution of read phase duration for each setup. Table 5.1 provides numerical comparison of the histograms shown in Figure 5.1. It can be seen that the centralised version has a majority of phases with the duration in between 0 and 20 milliseconds. The usual duration of the read phases for distributed installations is between 25 to 100 milliseconds. In

| | 1 Node | 2 Node | 3 Node | 4 Node | 5 Node |
|---|---|---|---|---|---|
| Average | 25.12 ms | 168.40 ms | 116.12 ms | 102.00 ms | 95.17 ms |
| Median | 9.00 ms | 125.00 ms | 62.00 ms | 58.00 ms | 59.00 ms |
| Mode | 8.00 ms | 24.00 ms | 24.00 ms | 25.00 ms | 33.00 ms |

Tabela 5.1: Numercial comparison of histogram shown in figure 5.1.

addition, there is a visible fraction of 12 to 4 % of phases for 2 to 5 node setups, respectively, taking longer than 300 milliseconds. This is an unfavourable result for the distributed installations of the system and calls for further examination.

To explain the increased duration of the read phases we show the observed amount of overhead originating from the synchronisation of nodes. Figure 5.2(a) contains a distribution of amounts of synchronisation done aggregated per system configuration. Figure 5.2(b) then shows differences between single installations. The process of synchronisation usually took 20 to 72 ms. It is also possible to see the increase of synchronisation overhead caused by the addition of more nodes.

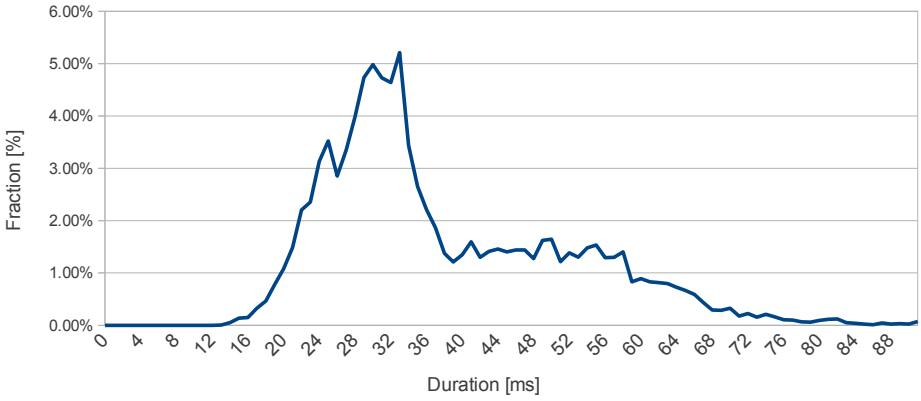
Due to the fact that this communication is necessary regardless the number of requests served during the given phase, all phases are inevitably burdened by the overhead (this includes also phases which process no requests and only perform the synchronisation), thus effectively rendering the read phases longer than phases of one node installation.

This observation, however, does not provide any information about the impact of the concept of migrating computation on the read phase durations. In order to investigate this aspect, we have removed the synchronisation time from the measured phase durations and obtained distribution shown in Figure 5.3.

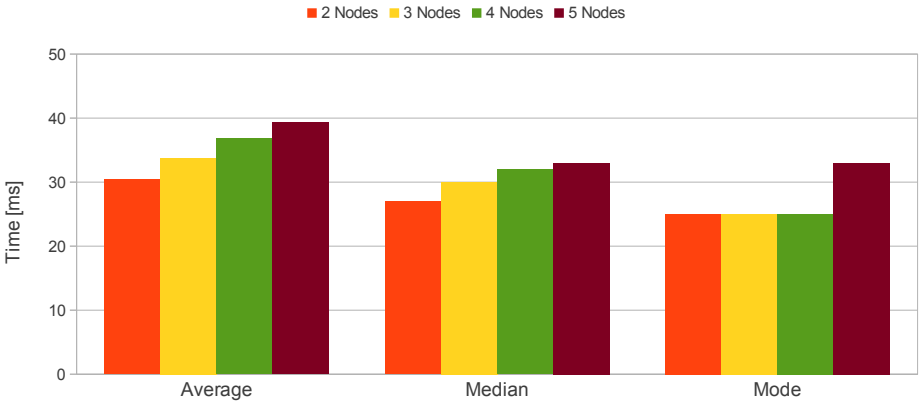
For one node system, approximately 82% of read phases took below 24 milliseconds. For 5-,4- and 3- node setups it was 62%, 56% and 50%, respectively. 2-node set-up had majority of phases taking longer than 24 milliseconds to complete.

However, there is still a substantial fraction of phases taking longer than 24 milliseconds. Figure 5.4 in Appendix A further describes this category. It can be seen that the read phase duration is steadily decreasing for all setups and that the range of 25 to 420 milliseconds exhaustively covers the distribution of read phase duration for all setups.

The final comparison of the read phases can be found in Figure 5.5. It can be concluded that:




(a) Distribution of synchronisation overhead.




(b) Comparison of synchronisation overhead for different node counts.

Figura 5.2: Details of synchronisation overhead.

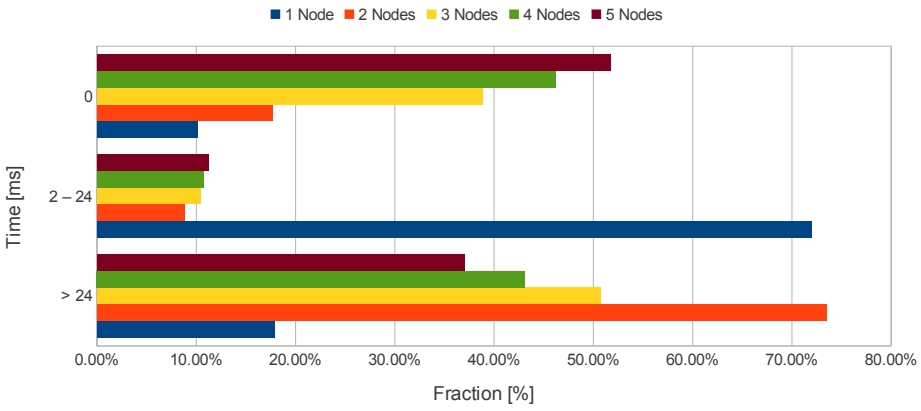



Figura 5.3: Read phase durations without synchronisation overhead.

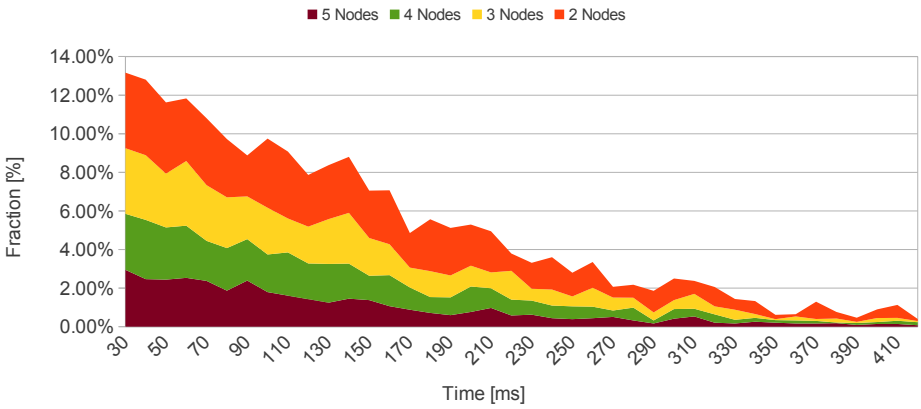



Figura 5.4: Read phase durations greater than 24 ms excluding synchronisation overhead.

- Synchronisation overhead has substantial impact on read phase duration.
- The overhead originating from migrating the computation increases the average read phase duration by 120 to 60 ms for 2-node to 5-node installation respectively, while keeping the median and mode favourable for the distributed installation for the system.
- The increased probability of having to migrate the computation, as the number of nodes increases, does not increase the read phase duration.

### 5.3.2 Write Phase Analysis

All the results presented in this section exclude time spent in synchronisation. We have shown in the preceding section this overhead is invariant, and removing it from the measurements allows gain clearer insight into the performance of the remaining system.

During the process of loading 400,000 terms into the system, we observed write phases of the durations summarised in Figure 5.6 and Table 5.2. It can be seen the 1-node system handled most write phases in 72 to 96 ms. The 2-node system handled most phases in 120 to 156 ms. 3- and 4- node systems have shown comparable distribution ranging from 10 to 130 ms with a minor increase around 60 ms. The 5-node system has a distribution peak around 4 to 6 ms, while the most write phases took 60 to 110 ms. There is also a notable local maximum of 2 ms long phases for 1-node system and visible fraction of phases taking longer than 230 ms.

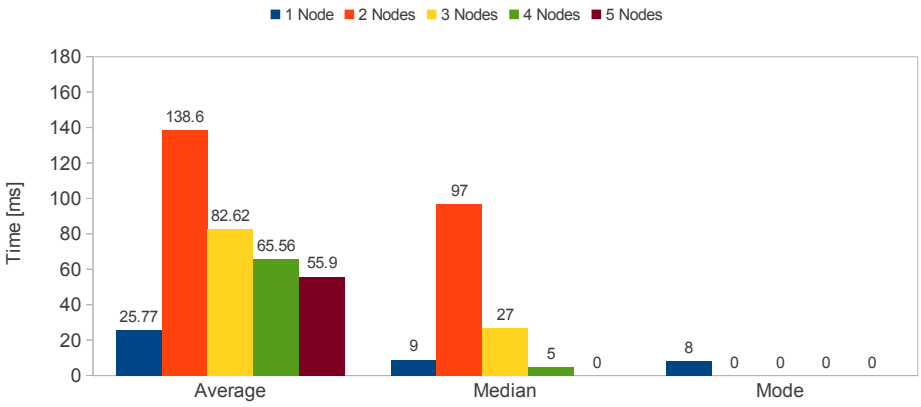



Figura 5.5: Read phase durations without synchronisation overhead.

| | 1 Node | 2 Node | 3 Node | 4 Node | 5 Node |
|---|---|---|---|---|---|
| Average | 192.54 | 178.11 | 96.14 | 90.39 | 78.83 |
| Median | 78 | 141 | 89 | 73 | 77 |
| Mode | 73 | 135 | 119 | 55 | 3 |

Tabela 5.2: Numerical comparison of write phase durations.

Table 5.2 also indicates variance of observed phase durations. One possible explanation of this variance can be the netwok glitches prolonging the phase durations by various amount of time, but not exhibiting any explainable distribution.

In Figure 5.6 we have also seen a portion of phases taking substantially longer time. Figure 5.7 shows these phases in greater detail. It can be seen that for the distributed system configurations, majority of phases took below 800 ms, while the centralised system suffered 50% of its long write phases being even longer than 800 ms.

## 5.4 Scalability

We will demonstrate the observed scalability of the proposed system by assesing the following:

1. Dividing computation-intensive operation amongst nodes.

2. Increased throughput.

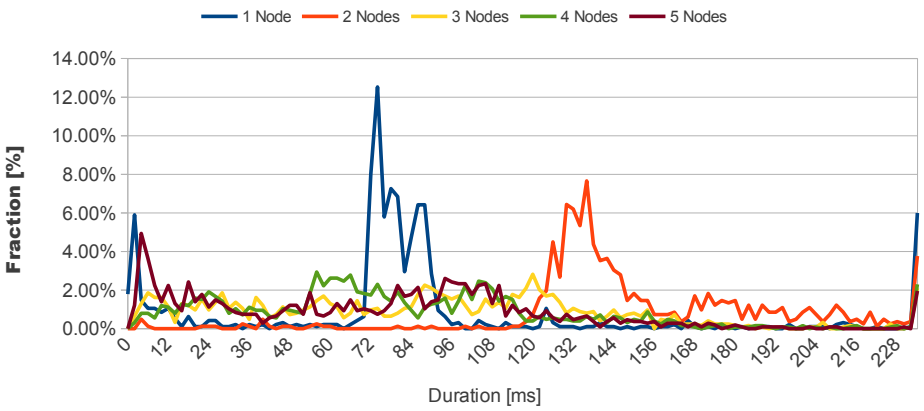


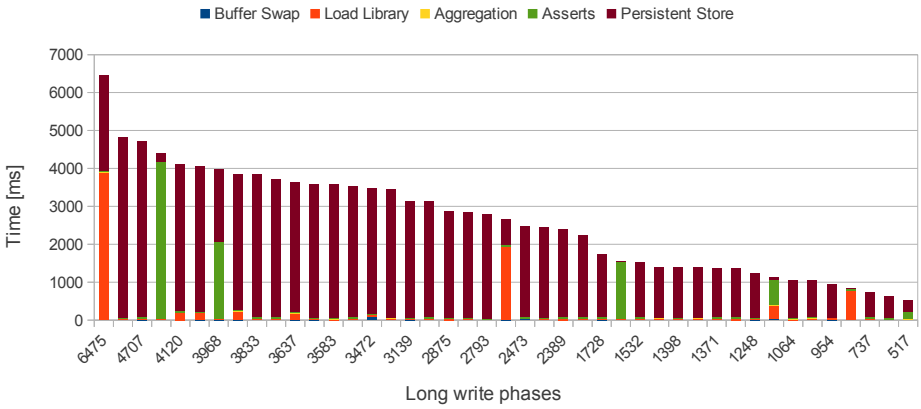

Figura 5.6: Distribution of write phase durations.

### 5.4.1 Computation Sharing

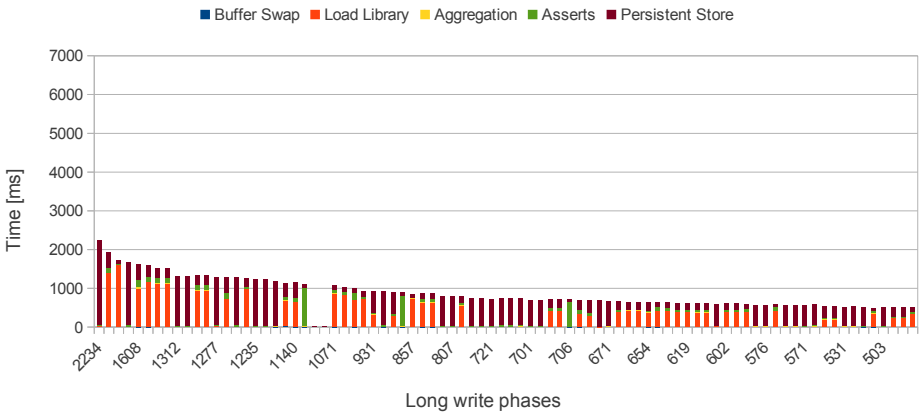
In Section 5.3.2 we analysed write phase durations. Write phases can be further split into following operations:

- *Buffer Swap.* Amount of time needed to process the buffer of incoming requests.
- *Library Load.* Time needed to load libraries from persistent storage into the Prolog engine.
- *Aggregation.* Time needed to group incoming requests into groups according to their name/arity. This operation was necessary to improve persistent storage performance.
- *Asserts.* Time it took to actually assert incoming therms into Prolog engine.
- *Persistent Store.* Time it took to store the terms into the persistent storage.

Figure 5.8 shows the average durations of each of the aforementioned operations executed during a write phase. It is clear that storing the incoming terms persistently took the largest portion of time. 1- and 2-node took similarly average time of 110 ms for this operation. For 3-,4- and 5- node systems this operation took averagely 60, 58 and 50 ms respectively. This improvement can be contributed to the performance of the underlying distributed storage solution when operating with 3 and more nodes. The second and third most time consuming operations are asserts and library loads taking between 40 and 18 ms for 1-node and 5-node respectively.




(a) Write Phases Longer Than 230 ms for 1-node installations.




(b) Write Phases Longer Than 230 ms for 2-5 node installations.

Figura 5.7: Examples of write phases lasting longer than 230 ms.

Figure 5.8 also shows how the average operation time decreases as the number of nodes is increasing. This can be explained by the fact the workload was constant, and that the nodes are splitting the computation needed to be done in order to handle the incoming write requests. This is also the first manifestation of the scalability of our system.

### 5.4.2 Increased Throughput

Our initial motivation was to build a system that can handle workload exceeding the limits of a conventional system. In order to evaluate if our system has this ability, we performed another set of experiments with the larger (1.5M terms) data set.

First we attempted loading the dataset into the system using different inbound throughputs.

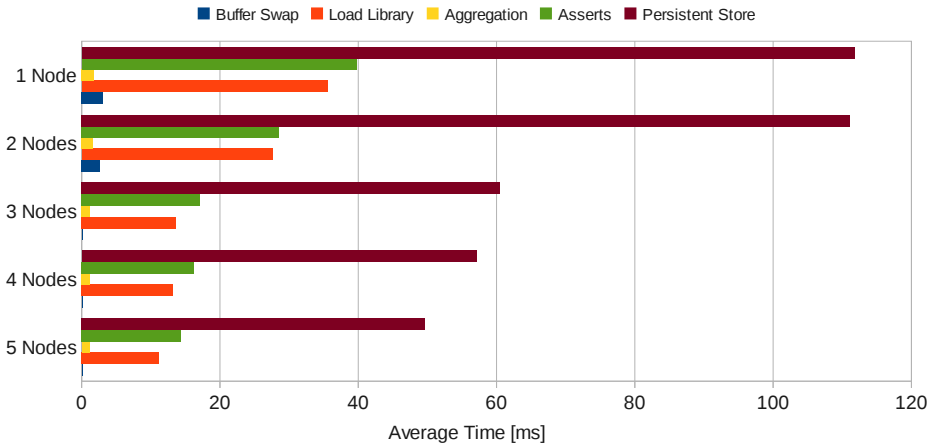
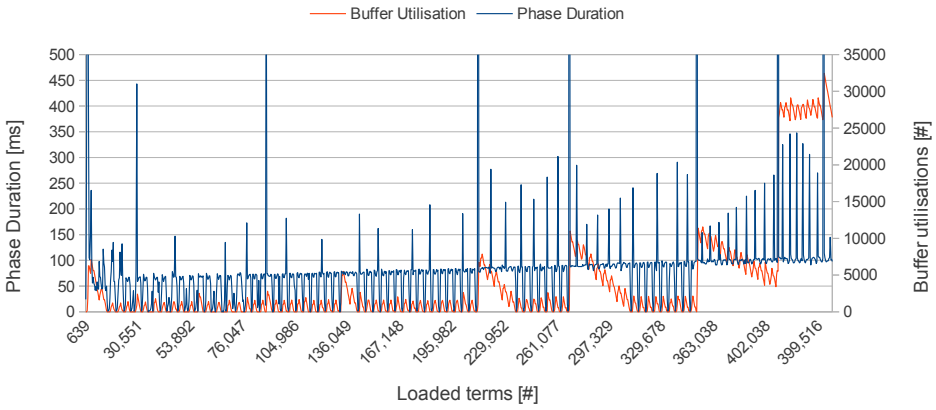



Figura 5.8: Decomposition of write phase duration into separate tasks.

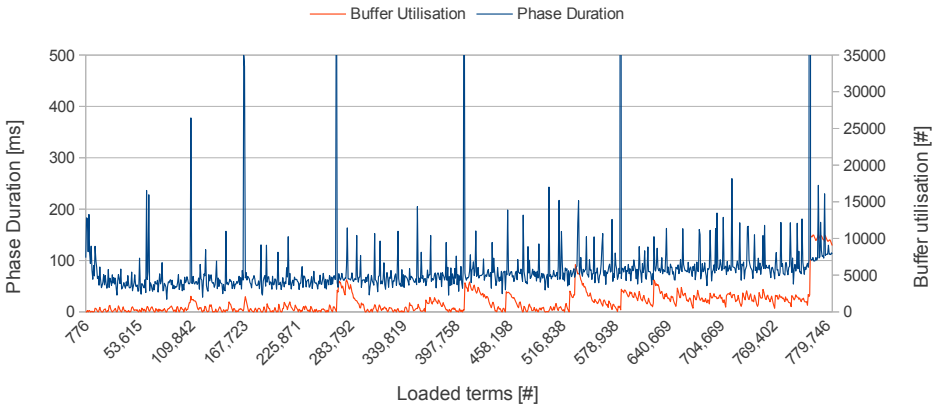
Figure 5.9 shows measured behaviour. The conclusions from these experiments are:

- 2-node system is already able to better handle the same throughput as the 1-node system
- 1-, 2- and 3-node systems crash once the amount of loaded terms in each node exceeds total capacity of each node, but 2- and 3-node systems can hold twice and trice more terms, respectively, than the 1-node system
- 4- and 5- node systems are able to hold 1,5M terms as expected
- 5-node system shows better performance than 4-node system in terms of phase duration being below 100 ms due to less terms being placed on each node.
- 5-node system is able to handle excessive throughput provided the excess is only temporary.
- Constant excessive throughput is not sustainable as it would lead to inbound request buffer overflow.
- The operations of all observed system exhibits occasional spikes that can be attributed to Java's garbage collection.

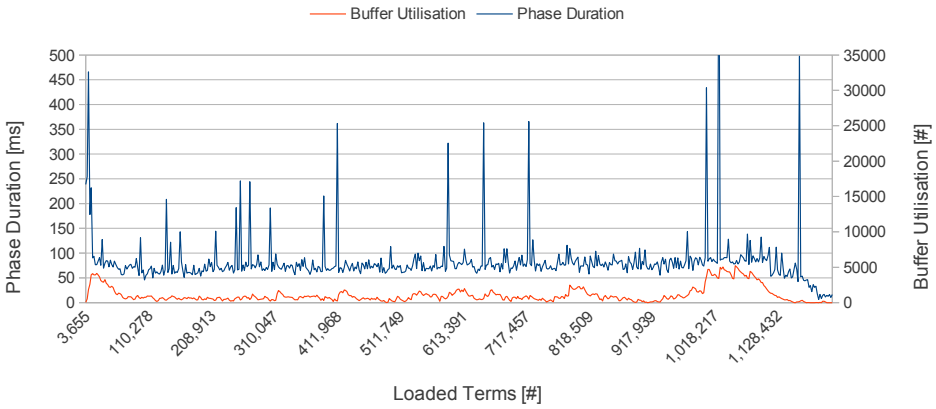
Once we have loaded 1.5M terms into the 5-node system, we initiated the consequent test of observing how the system continues its operation. We loaded the system by inbound traffic of 4K operations per second consisting of equal distribution of reads and writes. Figure 5.10 shows




(a) 1-Node system 4,000 op/s.




(b) 2-Node system 4,000 op/s.




(c) 5-Node system 9,000 op/s.

Figura 5.9: Increased throughput of different node configurations.

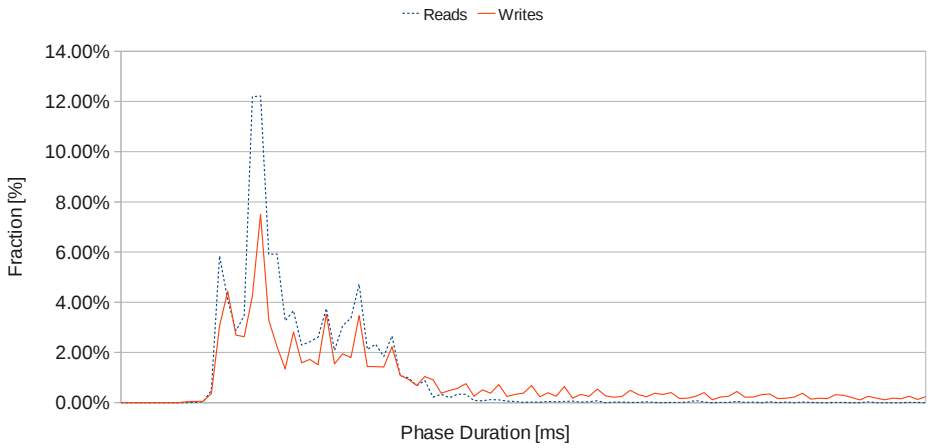



Figura 5.10: Operation of the system under heavy load.

distribution of phase durations. It can be seen that despite the rather heavy load, the system continues to operate with phase durations mostly influenced by the synchronisation overhead explained in Ssection 5.3.

## 5.5 Load Distribution

The final aspect we evaluate in this work is the question of load distribution. We have used two different data sets. Figure 5.11 shows how were the operations based on this data sets distributed among the nodes in the system. The figure shows the load is distributed uniformly.

The exception are 2- and 3- node installations handling 1.5M data set. Because the input terms were written in groups that did not follow the uniform distribution visible in 4- and 5-node installations. Because the system ran out of memory, the groups that would balance the load did not get written into the system.

## Summary

In this chapter we evaluated the prototype of our proposed system. At first, we explained the evaluation methodology being based on assessing distributed system against comparable centralised one. Then we examined what is the impact of overhead originating from the distributed

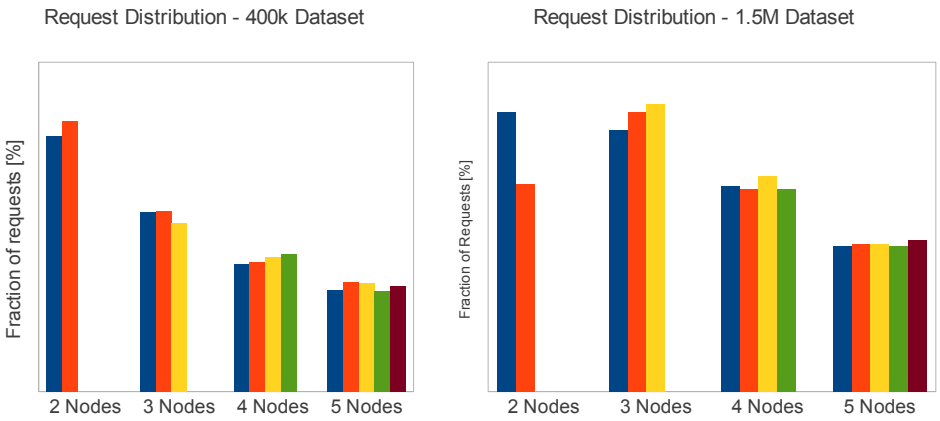



Figura 5.11: Request distribution per node for both used data sets.

aspects of the system. Then we demonstrated scalability and uniform load distribution.

The observer results can be summarised as follows:

- Synchronisation is the main source of overhead, burdening every phase of the system with 25-110 ms of delay.
- Migrating computation causes tolerable delay.
- The proposed system can handle more load than the centralised system thanks to balanced load distribution and consequent work-sharing.
- The proposed system effectively scales with more nodes being added.

# 6 Conclusions and Future Work

## 6.1 Conclusions

In this thesis we attempted to develop a prototype of a conceptual system utilising reasoning to facilitate application-level communication of devices connected to the Internet. The fundamental requirements for the system where the capacity to handle large knowledge bases and high amount of concurrent clients. We tried to meet these requirements by designing and implementing a prototype of ontology-based middleware offering distributed storage of the knowledge base, and decentralised and collaborative execution of the reasoning algorithm.

Our approach was based on the idea of using strings obtained by concatenation of names and arities of Prolog terms as the input for the hashing function of the underlying DHT implementation. This approach is able to split potentially large Prolog KB into disjoint parts while keeping terms of same name and arity clustered. Grouping of terms by their names and arities is native to Prolog, therefore this way of storage is well aligned with Prolog's computation model.

As a consequence of the storage approach, the complete solution of one goal requires involving multiple nodes in the computation. Here is where the second fundamental idea of this thesis lies. Instead of transferring the terms needed for the computation we let the computation "travel"to the terms, assuming the computation context needed to be transmitted is smaller and therefore easier to be transmitted than the required data. The benefit resulting from this computational model lies in gaining the ability to compute other solutions while not working on the original one, thus allowing the execution of more concurrent solutions.

The designed architecture assumes phase-based operation that requires node synchronisation. The employed synchronisation mechanisms cause the distributed system to be about 50 ms slower than the centralised system on a per-phase basis. The concept of migrating computation causes overhead of averagely 138 ms to 56 ms for 2-node to 5-node installations of the system respectively, therefore refuting the assumption that increased probability of migrating the

computation would lead to increased solution durations. However, the write operations are not burdened with the migration overhead and therefore exhibit faster execution times. Moreover, the experiments also demonstrated scalability of the system in terms of its ability to handle more inbound operations and larger volumes of data, when increasing number of nodes participating in the system, while maintaining desired interactive responsiveness.

## 6.2 Future Work

The possible directions of the future work are numerous.

Parallel Prolog implementations, such as SICStus (Carlsson, Widen, Andersson, Andersson, Boortz, Nilsson, & Sjöland 1988), offer different level of parallelism and concurrency. Therefore, it can be investigated what are the ways of merging this approach with the one proposed in this thesis. The main difficulty would lie in the fact that migrating computation often implies interrupting the computation, what could effectively prevent any benefit from parallelizing the solution within one node.

Another possible direction is extending the read phase functionality by caching of (partial) results. This could improve the overall performance as it would limit the need of repeated goal solving. Cached items can be used for implementing fault-tolerance as well. However, the main problem would be maintaining the guarantee of obtaining consistent query solutions.

The current approach to ensuring consistency could also be revisited. The phase-based operation requires excessive synchronisation. Finding a way of ensuring the consistency without the synchronisation would remove the inevitable synchronisation overhead observed during the evaluation of the proposed system.

Finally, it is also possible to improve the choice of input for the hashing function. With a more sophisticated approach, it would be possible to store not only terms of the same name/arity, but also terms that are close to each other in a Prolog rule at the same node. This would remove the need to migrate the computation, especially if multiple hashing inputs would be combined and used together, but has the potential to threaten the uniform load distribution.

# Referências


Anadiotis, G., S. Kotoulas, & R. Siebes (2007). An architecture for peer-to-peer reasoning. In *Proc. of the First Int. Workshop"New forms of reasoning for the Semantic Web: scalable, tolerant and dynamic", co-located with ISWC 2007 and ASWC 2007*, Volume 291.

Baldauf, M., S. Dustdar, & F. Rosenberg (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing 2*(4), 263–277.

Bandyopadhyay, S., M. Sengupta, S. Maiti, & S. Dutta (2011). Role of middleware for internet of things: A study. *International Journal of Computer Science & Engineering Survey (IJCSES) 2*(3), 94–105.

Bettini, C., O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, & D. Riboni (2010). A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing 6*(2), 161 – 180.

Brachman, R. & H. Levesque (2004). *Knowledge representation and reasoning*. Morgan Kaufmann.

Carlsson, M., J. Widen, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, & T. Sjöland (1988). *SICStus Prolog user's manual*, Volume 3. Swedish Institute of Computer Science.

Dean, J. & S. Ghemawat (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*(1), 107–113.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, & W. Vogels (2007). Dynamo: amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Volume 14, pp. 205–220.

Ericsson (2011, February). More than 50 billion connected devices. `http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf`. Whitepaper.

Fang, Q., Y. Zhao, G. Yang, & W. Zheng (2008). Scalable distributed ontology reasoning using dht-based partitioning. In *The Semantic Web*, pp. 91–105. Springer.

Gilbert, S. & N. Lynch (2002, June). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*(2), 51–59.

Gómez-Goiri, A. & D. López-De-Ipiña (2010). A triple space-based semantic distributed middleware for internet of things. In *Proceedings of the 10th international conference on Current trends in web engineering*, ICWE'10, Berlin, Heidelberg, pp. 447–458. Springer-Verlag.

Gu, T., X. H. Wang, H. K. Pung, & D. Q. Zhang (2004). An ontology-based context model in intelligent environments. In *Proceedings of communication networks and distributed systems modeling and simulation conference*, Volume 2004, pp. 270–275.

Gupta, G., E. Pontelli, K. A. Ali, M. Carlsson, & M. V. Hermenegildo (2001, July). Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst. 23*(4), 472–602.

Horn, A. (1951). On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic 16*(1), 14–21.

Hunt, P., M. Konar, F. P. Junqueira, & B. Reed (2010). Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, Volume 8, pp. 11–11.

Jackson, P. (1990). *Introduction to expert systems*. Addison-Wesley Longman Publishing Co., Inc.

Karger, D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine, & D. Lewin (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663. ACM.

Kiryakov, A., K. I. Simov, & D. Ognyanov (2002). Ontology middleware: Analysis and design. *On-To-Knowledge (IST-1999-10132) Deliverable 38*.

Knuth, D. E. (2006). *The art of computer programming. 4, fascicle 4, 1. print.. Generating all trees*. addison-Wesley.

Kowalski, R. (1973). *Predicate logic as programming language*. Edinburgh University.

Lakshman, A. & P. Malik (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*(2), 35–40.

Lamport, L. (2001). Paxos made simple. *ACM SIGACT News 32*(4), 18–25.

Lassila, O., R. R. Swick, W. Wide, & W. Consortium (1998). Resource description framework (rdf) model and syntax specification.

Lua, E. K., J. Crowcroft, M. Pias, R. Sharma, & S. Lim (2005). A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials 7*(2), 72–93.

Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, & G. Czajkowski (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146. ACM.

Matuszek, C., J. Cabral, M. Witbrock, & J. DeOliveira (2006). An introduction to the syntax and content of cyc. In *Proceedings of the 2006 AAAI spring symposium on formalizing and compiling background knowledge and its applications to knowledge representation and question answering*, Volume 3864. Citeseer.

McGuinness, D. L., F. Van Harmelen, et al. (2004). Owl web ontology language overview. *W3C recommendation 10*(2004-03), 10.

Nilsson, N. J. (1986). Probabilistic logic. *Artificial intelligence 28*(1), 71–87.

Nixon, L. J., E. Simperl, R. Krummenacher, & F. Martin-Recuerda (2008). Tuplespace-based computing for the semantic web: A survey of the state-of-the-art. *Knowledge Engineering Review 23*(2), 181–212.

Piancastelli, G., A. Benini, A. Omicini, & A. Ricci (2008, 16–20 March). The architecture and design of a malleable object-oriented Prolog engine. In R. L. Wainwright, H. M. Haddad, R. Menezes, & M. Viroli (Eds.), *23rd ACM Symposium on Applied Computing (SAC 2008)*, Volume 1, Fortaleza, Ceará, Brazil, pp. 191–197. ACM. Special Track on Programming Languages.

Qin, W., Y. Shi, & Y. Suo (2007). Ontology-based context-aware middleware for smart spaces. *Tsinghua Science & Technology 12*(6), 707–713.

Ratnasamy, S., P. Francis, M. Handley, R. Karp, & S. Shenker (2001). *A scalable content-addressable network*, Volume 31. ACM.

Rowstron, A. & P. Druschel (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pp. 329–350. Springer.

Russell, S. J., P. Norvig, J. F. Canny, J. M. Malik, & D. D. Edwards (1995). *Artificial intelligence: a modern approach*, Volume 74. Prentice hall Englewood Cliffs.

Satyanarayanan, M. (2001). Pervasive computing: Vision and challenges. *Personal Communications, IEEE 8*(4), 10–17.

Schilit, B. N. & U. Sengupta (2004). Device ensembles [ubiquitous computing]. *Computer 37*(12), 56–64.

Serafini, L. & A. Tamilin (2005). Drago: Distributed reasoning architecture for the semantic web. In *The Semantic Web: Research and Applications*, pp. 361–376. Springer.

Song, Z., A. A. Cárdenas, & R. Masuoka (2010). Semantic middleware for the internet of things. In *Internet of Things (IOT), 2010*, pp. 1–8. IEEE.

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*, Volume 31, pp. 149–160. ACM.

Teixeira, T., S. Hachem, V. Issarny, & N. Georgantas (2011). Service oriented middleware for the internet of things: a perspective. In *Proceedings of the 4th European conference on Towards a service-based internet*, ServiceWave'11, Berlin, Heidelberg, pp. 220–229. Springer-Verlag.

Urbani, J., S. Kotoulas, E. Oren, & F. Van Harmelen (2009). Scalable distributed reasoning using mapreduce. In *The Semantic Web-ISWC 2009*, pp. 634–649. Springer.

Wang, C. & B. Li (2003). Peer-to-peer overlay networks: A survey. *Department of Computer Science, The Hong Kong University of Science and Technology*.

Weiser, M. (1993, July). Some computer science issues in ubiquitous computing. *Commun. ACM 36*(7), 75–84.

Zhao, B. Y., J. Kubiatowicz, A. D. Joseph, et al. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and routing.