FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Enhance Service-Oriented Architecture in Cyber-Physical Production System Design

Vinicius Werlich Kuci Barbosa



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Rui Pedro Ferreira Pinto

October 30, 2026

Resumo

O avanço dos conceitos da Indústria 4.0 tem impulsionado a demanda por sistemas de automação industrial mais adaptáveis, modulares e flexíveis. Sistemas Ciberfísicos de Produção (CPPS) devem permitir reconfigurações frequentes, integração transparente de recursos heterogêneos e funcionalidades avançadas orientadas por dados, ao mesmo tempo em que atendem a requisitos industriais específicos, como desempenho em tempo real e confiabilidade. No entanto, muitas das arquiteturas de controlo industrial atualmente empregadas ainda dependem de configurações estáticas em alguma das etapas de projeto ou implantação, o que limita sua capacidade de adaptação às constantes mudanças das demandas de produção.

Este trabalho investiga estratégias arquiteturais a fim de aumentar a modularidade e a flexibilidade em tempo de execução de CPPS. Com o objetivo de viabilizar recursos *plug-and-produce*, é adotada uma abordagem baseada em serviços, na qual é proposto um mecanismo distribuído de descoberta semântica em duas fases, construído sobre protocolos padronizados e sem a necessidade de uma infraestrutura centralizada de registro.

O mecanismo é validado em um ambiente de testes composto por dispositivos conteinerizados distribuídos em uma rede física e avaliado, sob as mesmas condições experimentais, em comparação com o mecanismo de descoberta amplamente utilizado em Arquiteturas Orientadas a Serviços (*SOA*) industriais. Além disso, uma validação final confirmou que a solução desenvolvida é capaz de descobrir e estabelecer ligação com um serviço remoto a partir de uma aplicação industrial em conformidade com a norma da *International Electrotechnical Commission* (*IEC*) 61499.

A abordagem proposta, em contraste com o mechanismos legado, foi capaz de filtrar semanticamente de forma eficiente serviços que se anunciaram na rede via multicast. Como resultado, o custo da arquitetura proposta, em termos de processamento, se mostrou cerca de $8\times$ mais eficiente do que a arquitetura legado por cada resposta processada, ao mesmo tempo que reduz o tráfego de descoberta em cerca de 84%, um fator de 6,4 numa população de dezasseis dispositivos em que apenas um satisfaz o pedido. A latência total da descoberta é predominantemente determinada pelos atrasos impostos pelos protocolos, e não pelo tempo de processamento, indicando que a abordagem é adequada para dispositivos com recursos computacionais limitados. Os resultados demonstram que realizar a filtragem semântica inicial no nível da rede possibilita uma descoberta distribuída eficiente, preservando a expressividade semântica e mantendo baixo o custo de comunicação.

Palavras-chave: Sistemas de Produção Ciber-Físicos; Arquitetura Orientada a Serviços; Descoberta de Serviços; Plug-and-Produce; IEC 61499; mDNS; CoAP; Descrição Semântica.

Abstract

The increased advance of Industry 4.0 concepts has led to a growing demand for adaptable, modular, and flexible industrial automation systems. Cyber-Physical Production Systems (CPPS) must allow for frequent reconfigurations, seamless integration of heterogeneous resources, and advanced data-driven functionalities, while meeting specific industrial requirements, such as real-time performance and reliability. However, many of the industrial control architectures used today are still inflexible and dependent on static configurations at some point of the deployment and design, which restricts their ability to flexibly adapt to constantly changing production demands.

This body of work investigates architectural strategies that enhance the modularity and execution time flexibility of CPPS. In order to provide plug-and-produce resources, a service-based design is approached, and a two-phase distributed semantic discovery mechanism is proposed, built on standardized protocols and requiring no central registry infrastructure.

The mechanism is validated on a testbed of containerized devices deployed over a physical network and evaluated, under the same experimental conditions, against the discovery mechanism widely adopted in industrial Service-Oriented Architectures (SOA). In addition, a final validation confirmed that the proposed solution is capable of discovering and binding to a remote service from an industrial application compliant with the International Electrotechnical Commission (IEC) 61499 standard. In contrast to the baseline mechanism, the proposed approach was able to efficiently perform semantic filtering of services advertised through multicast. As a result, the proposed architecture required approximately eight times less processing per response than the baseline while reducing discovery traffic by about 84%, a factor of 6.4 in a network of sixteen devices where only one satisfied the request. Overall discovery latency is dominated by protocol-imposed delays rather than processing time, indicating that the approach is suitable for constrained devices. The results show that performing the initial semantic filtering at the network level enables efficient distributed discovery while preserving semantic expressiveness and maintaining low communication overhead.

Keywords: Cyber-Physical Production Systems; Service-Oriented Architecture; Service Discovery; Plug-and-Produce; IEC 61499; mDNS; CoAP; Semantic Description.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. This dissertation contributes primarily to SDG 9 by proposing a distributed service discovery mechanism that improves the flexibility, interoperability and scalability of industrial automation systems. It also contributes to SDG 12 by supporting more efficient use of industrial resources through dynamic service discovery and runtime reconfiguration.

The specific Sustainable Development Goals addressed by this dissertation are:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

SDG 12 Ensure sustainable consumption and production patterns.

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.5	Develops a distributed semantic service discovery mechanism that promotes innovation in industrial automation by enabling runtime reconfiguration and interoperability between heterogeneous devices.	Discovery latency; communication overhead; CPU utilization; scalability with network population.
	9.c	Promotes modern industrial communication infrastructure through standards-based, decentralized service discovery without requiring centralized registries.	Number of responding devices; network traffic per discovery; successful runtime service bindings.
12	12.2	Improves the utilization of industrial resources by allowing applications to dynamically discover and bind only to devices capable of providing the requested functionality, avoiding unnecessary communication and supporting resource reuse.	Reduction in unnecessary discovery traffic; successful semantic filtering; network efficiency.

Acknowledgements

Esta dissertação é o produto de uma jornada que durou seis anos, e que não se fez sozinha. Entre momentos de plenitude e de dificuldade, agradeço às pessoas que estiveram comigo em todos eles.

Aos meus pais, Salete Werlich e Fabricio Kuci Barbosa, que mesmo a um oceano de distância me ofereceram todo o apoio possível para que eu chegasse onde cheguei. Pelo incentivo e suporte que me deram para sempre buscar conhecimento e que, mesmo de longe, ainda o fazem por hábito.

À minha família, que pelos mesmos motivos compôs os bastidores durante a formação do carácter de que tenho tanto orgulho.

Ao Professor Rui Pinto, que se manteve em contacto comigo desde a minha licenciatura e que, mesmo com o conhecimento da minha condição de estudante trabalhador, confiou um trabalho de tal importância a mim.

Aos meus amigos "Cristiners" e ex-colegas de casa Karla, Gustavo, Júlia, Tatiana e Gabriela, que se tornaram a minha família em território estrangeiro e que me proporcionaram momentos inesquecíveis desde que cheguei a Portugal. Da mesma forma agradeço aos amigos que, mesmo à distância, se mostram presentes em minha vida.

Aos meus colegas de trabalho, que durante o último ano acompanharam a jornada que foi a composição deste documento e que tanto me ensinaram profissionalmente.

Aos meus amigos batuqueiros, que nos meus momentos de esgotamento foram o meu alívio e que têm por si um lugar tão especial na minha vida.

Por fim, agradeço à comunidade portuguesa e à comunidade académica da Universidade do Porto, que me proporcionaram viver todos estes momentos.

Vinicius Werlich Kuci Barbosa

"Imagination is more important than knowledge."

Albert Einstein

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Objectives . . . 4
- 1.4 Contributions . . . 4
- 1.5 Document Structure . . . 5

2 Background knowledge **6**
- 2.1 The Industry 4.0 and Cyber-Physical Production Systems . . . 6
- 2.2 IEC 61499 Standard . . . 7
- 2.3 Service Oriented Architecture . . . 9
  - 2.3.1 Key Principles of SOA . . . 10
  - 2.3.2 Synchronous and Asynchronous Interaction Models . . . 11
- 2.4 Legacy Web Services and DPWS . . . 11
  - 2.4.1 Web Services for Industrial Embedded Devices . . . 12
- 2.5 RESTful Web Services and Constrained Protocols . . . 13
  - 2.5.1 CoAP Ecosystem . . . 14
- 2.6 Service Discovery Mechanisms . . . 14
  - 2.6.1 Centralized Registry-based Discovery Architecture . . . 14
  - 2.6.2 Distributed Discovery Architecture . . . 15

3 State of the Art **17**
- 3.1 Service-Oriented Architectures in Cyber Production Systems . . . 17
  - 3.1.1 Service Discovery and Composition . . . 21
  - 3.1.2 Semantic Description and Matchmaking . . . 24
- 3.2 Relevant Contributions from IT . . . 25
- 3.3 Conclusion . . . 27

4 Proposed Solution **30**
- 4.1 Rationale and Requirements . . . 31
- 4.2 Architectural Overview . . . 32
- 4.3 Phase 1: Semantic Pre-filtering . . . 34
  - 4.3.1 mDNS Announcement Model . . . 34
  - 4.3.2 TXT Record Metadata Structure . . . 36
  - 4.3.3 Pre-filter Evaluation . . . 37
- 4.4 Phase 2: CoAP Semantic Retrieval and Matchmaking . . . 38
  - 4.4.1 Candidate Ranking . . . 38
  - 4.4.2 CoAP Semantic Metadata Document . . . 40

4.4.3 Semantic Matchmaking . . . 43
4.5 Service Binding and Caching . . . 43
4.5.1 Service Binding . . . 44
4.5.2 Cache Model . . . 44

5 Implementation **46**
5.1 Overview . . . 46
5.1.1 Chosen Language . . . 47
5.1.2 Libraries . . . 47
5.2 Anatomy of a Discovery . . . 48
5.2.1 Device Configuration . . . 49
5.2.2 The Provider . . . 49
5.2.3 The Search Engine . . . 51
5.3 Decisions Forced by the Implementation . . . 52

6 Validation **54**
6.1 Methodology . . . 54
6.1.1 Structure of the Validation . . . 55
6.2 Measurement and Qualification of the Architecture . . . 56
6.2.1 Properties of the Environment Affecting Measurement . . . 57
6.2.2 A Descriptive Model of the Overhead . . . 57
6.3 Testbed Design . . . 58
6.3.1 Execution Environment . . . 58
6.3.2 The Reference Baseline . . . 59
6.4 Binding Inside an IEC 61499 Runtime . . . 60
6.4.1 The RTE . . . 60
6.4.2 The Application . . . 61
6.4.3 Instantiating the Discovered Block at Runtime . . . 63

7 Results **66**
7.1 Prior Network Segment Characterisation . . . 66
7.2 Correctness . . . 68
7.3 Performance . . . 69
7.3.1 Network Overhead . . . 69
7.3.2 Waiting Against Computation . . . 71
7.4 Scalability . . . 74
7.4.1 The Overhead Model Instantiated . . . 75
7.5 Compliance . . . 77
7.5.1 The Cycle Exercised . . . 78
7.5.2 Candidate Evaluation . . . 79
7.5.3 Instantiating the Discovered Block at Runtime . . . 79

8 Discussion **81**
8.1 Intrinsic Behavior of the Protocols . . . 81
8.2 Answering the Research Questions . . . 82
8.2.1 Research Question 1 . . . 82
8.2.2 Research Question 2 . . . 84
8.3 What the Work Does Not Establish . . . 85
8.3.1 Limitations of the Method . . . 85

8.3.2 Limitations of the Implementation 86
8.3.3 Limitations of the Design 87

9 Conclusion **88**
9.1 Contributions 88
9.2 Achievement of the Objectives 89
9.3 The Target Environments 90
9.4 Scientific Impact 91
9.5 Future Work 91

References **94**

List of Figures

2.1 IEC 61499 application deployment and distribution. From [51]. 8
2.2 Representation of service messaging at different locations. 10
2.3 Simplified layered version of the DPWS protocol stack. 13

3.1 Service Oriented infrastructure. From [7]. 18
3.2 Basic SOA architecture for IEC 61499 standard. From [13] 19
3.3 Layered structure of the DPWS protocol stack for IEC 61499 management. Adapted from [12]. 22

4.1 Overview of the two-phase semantic discovery architecture. Represents all the components required for the design implementation. 33
4.2 Structure of the proposed mDNS service query (PTR record). 35
4.3 Structure of a PTR response derived from a previous query. 35
4.4 Dissection of an mDNS response: Example of an additional section with expanded TXT record. 37
4.5 Class diagram of TD core vocabulary. From [30]. 41

5.1 Component view of the implementation. 48
5.2 Service provider's side class diagram. 50
5.3 Semantic Search Engine class diagram. 51

6.1 Testbed physical deployment. 59
6.2 Provider side FB system diagram. 61
6.3 Consumer side FB system diagram. 62
6.4 Sequence of one service invocation, from the request event raised by the consumer application to the result returned by the provider's FB network. 63
6.5 Semantic Deployer FB integrated with the Semantic Discover. 64
6.6 Semantic Deployer Sequence Diagram. 65

7.1 The funnel over the same population of sixteen devices, in the two extreme compositions. 68
7.2 Waiting against computation fraction between the proposed architecture and the baseline. Considering total discovery time. 71
7.3 Phase 1 waiting and computing against the number of responding providers. . . . 72
7.4 Phase 2 waiting and computing against the number of documents fetched. 73
7.5 Cost of evaluating one document, measured in isolation, against the per-document CPU measured during the campaign. 73
7.6 Payload received and marginal cost against the number of devices of the requested category. 74

7.7 The model of Equation 7.3 evaluated against the number of devices of the requested category. The shaded region is the range the campaign covered and the dashed continuation is the model 77

List of Tables

3.1 Comparison of the main service discovery approaches reviewed in this chapter. Full support (●), partial support (○), not addressed (–). 28

6.1 Runtime compatibility fixes applied to DINASORE for execution with recent Python versions and interoperability with 4DIAC. 61

7.1 The segment, characterised over the 60 seconds immediately before the campaign. 67
7.2 Cardinality of the three sets across forty repetitions of each composition. A single value means the same number was observed in every repetition while a range means variation. 68
7.3 Cost of the exchanges, in bytes, a discovery requires for a single provider, all with zero variance. 70
7.4 One complete discovery of the same service by the two systems, in bytes, for a single provider. The *effective* figures count only the packets required by the discovery protocol, whereas the *gross* figures include all traffic emitted during the probe window. 70
7.5 A population of sixteen devices, of which r belong to the requested category. Payload in bytes received, packets received, and first-phase CPU in milliseconds, as medians. 75
7.6 One complete cycle, in milliseconds, over the 40 samples that follow the discarded warm-up. The simulated duration of the remote welding job is fixed at 50 ms and is contained in t_{app}. 78
7.7 The cycle with the proxy block drawn in advance and with the proxy block created at runtime, in milliseconds, as medians over 41 samples in each campaign. 80

9.1 The four objectives against the evidence produced by this dissertation. 90

List of Acronyms

BFB	Basic Function Block
CFB	Composite Function Block
CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
CPMS	Cyber-Physical Manufacturing Systems
CPPS	Cyber-Physical Production Systems
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
DGT	Distributed Geographic Table
DHT	Distributed Hash Table
DINASORE	Dynamic Intelligent Architecture for Software and Modular Reconfiguration
DL	Description Logics
DLL	Dynamic-Link Library
DLS	Distributed Location Service
DNS	Domain Name System
DNS-SD	Domain Name System-Based Service Discovery
DPWS	Device Profile for Web Services
DTLS	Datagram Transport Layer Security
FB	Function Block
FBSRT	Function Block Service Runtime
GDS	Global Discovery Server
HMI	Human-Machine Interface
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force
IO	Input/Output
IoT	Internet of Things
IP	Internet Protocol
ISA	International Society of Automation
IT	Information Technology
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
LDS-ME	Local Discovery Server with Multicast Extension
LED	Light-Emitting Diode
LRD	Local Resource Directory
LRU	Least Recently Used

MASON	MAnufacturing's Semantics ONtology
mDNS	Multicast Domain Name System
OPC UA	Open Platform Communications Unified Architecture
OSI	Open Systems Interconnection
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
P2P	Peer-to-Peer
PLC	Programmable Logic Controller
PSL	Process Specification Language
PTR	Pointer Record
QoS	Quality of Service
RD	Resource Directory
RDF	Resource Description Framework
REST	Representational State Transfer
RFC	Request for Comments
RPC	Remote Procedure Call
RST	Reset
RTE	Runtime Environment
SDG	Sustainable Development Goal
SIFB	Service Interface Function Block
SIRENA	Service Infrastructure for Real-time Embedded Networked Applications
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SPARQL	SPARQL Protocol and RDF Query Language
SQWRL	Semantic Query-Enhanced Web Rule Language
SRV	Service Record
TCP	Transmission Control Protocol
TD	Thing Description
TTL	Time-to-Live
TXT	Text Record
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
VLAN	Virtual Local Area Network
VM	Virtual Machine
W3C	World Wide Web Consortium
WoT	Web of Things
WS	Web Services
WSDL	Web Services Description Language
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language
ZeroConf	Zero-Configuration

Chapter 1

Introduction

The advancement of industrial automation and the demand for autonomous and distributed systems have significantly accelerated the adoption of interdisciplinary paradigms in the past 30 years. Each conceptual revolution in industry, driven by technological contexts, has contributed to the convergence of new industrial system architectures, with solid concepts from different technological areas. Industry 4.0, the paradigm that frames this work, has propelled the realization of Cyber-Physical Systems (CPS) in an industrial context, and remains the reference under which distributed industrial control are currently designed and deployed as further discussed in Chapter 3.

Architectural concepts in information technology, such as Service-Oriented Architecture (SOA), are attracting the attention of the industrial automation community [7, 11]. In parallel, rapid changes in consumption and manufacturing patterns fundamentally require reconfiguration of the production pipeline with minimum downtime. In addition to this, industrial equipment is becoming even more intelligent and capable. In the context of an agent-based model widely adopted in modern industries, components of a manufacture pipeline are able to take decisions and communicate with each other through their virtual models [46]. Such behavior is strongly supported by the adoption of distributed systems and Plug-and-Produce concepts.

The growing trend of interconnected devices in the same context has motivated the elaboration of new standards and specifications, aiming to improve the portability and distribution of applications across heterogeneous platforms. A representative example is the International Electrotechnical Commission (IEC) 61499 standard [26], which separates application logic from the hardware resources on which it is executed.

Finally, this evolution establishes the architectural conditions for flexible and distributed industrial systems, but it does not by itself provide the mechanisms to take advantage of this flexibility at runtime. In particular, many implementations of the Plug-and-Produce model, despite their close relationship with SOA principles, do not fully address important requirements of modern Cyber-Physical Production Systems (CPPS), including semantic service descriptions, dynamic and distributed discovery and suitability for resource-constrained devices, service discovery thus

becomes the missing link between the architectural flexibility promised by modern industrial standards.

1.1 Motivation

The changes in the consumption patterns in the past few decades have imposed stringent requirements to modern production systems. Production lines are expected to support frequent reconfiguration without compromising the production throughput and, despite recent adoptions of standardized architectures, industrial environments still struggle with seamless integration of heterogeneous devices and Plug-and-Produce concepts. As a consequence, there is a growing need for an architectural approach that enables regular and dynamic expansion of a production system.

Although modular design of distributed industrial systems has been possible since the third industrial revolution and considering that recent industrial standards have significantly advanced the modularity and distribution of automation systems, important aspects of true plug-and-produce operation remain unresolved. The IEC 61499 standard is a representative example. By separating application logic from the underlying execution platform, the standard enables applications to be deployed and reconfigured independently of specific devices, providing an architectural foundation compatible with SOA principles [13, 26].

The fact that the standard has a close relation with SOA is the main reason why IEC 61499 is relevant for this dissertation. The distributed model and the separation of application and execution resources set up most of the conditions needed for runtime service composition. Yet, the standard does not specify how an application discovers new resources while it is in operation. This means the device mappings have to be obligatorily set up offline using the engineering tool and included in the deployment configuration.

Consequently, although the tight coupling between application logic and hardware can be reduced in an IEC 61499 environment, the binding between a service consumer and a service provider remains static. This prevents the level of dynamic composition and plug-and-produce behaviour expected from SOA industrial systems [7, 12, 16]. The architecture therefore performs most of the work required for distribution and reconfiguration, while leaving the binding that would activate this flexibility at runtime outside the scope of the standard.

This work is focused on IEC 61499 for both reasons. First, the standard exposes one clear gap that remains between the flexibility offered by modern distributed control architectures and the discovery mechanisms required to exploit that flexibility. Second, its close alignment with SOA makes it an appropriate context to investigate service discovery, semantic matching and runtime binding. The lack of these mechanisms is especially important in an architecture that already provides the conditions for a distributed service composition. Bridging this gap is therefore important to realise the modularity, adaptability and plug-and-produce operation envisioned by Industry 4.0.

1.2 Problem Statement

Despite the wide adoption of concepts like CPPS and standardized communication protocols, none of these concepts draws a unified architecture for autonomous service discovery and semantic matching. Production lines are still not capable of identifying bottlenecks and smoothly binding their processes to more capable resources during runtime, and the binding decision remains an engineering-time artefact.

For that binding to be made with no downtime, a consumer must be able to answer a question that no current mechanism answers completely in a distributed way:

> *"Which device on this network can perform this task, given all the constraints and requirements the task actually imposes?"*

Answering it demands four properties at once. The mechanism has to be *fully distributed*, because any registry that has to be configured before the system runs re-introduces the pre-configuration that plug-and-produce is meant to eliminate. It must be *lightweight*, in traffic, and in computation, as it may share a network, and often a processor, with control traffic on constrained equipment. It must be *semantically expressive* enough for a consumer to discriminate a device of the right type that is capable of the task from one that is not. And, at last, it has to be *deployable on existing infrastructure*, no extensions to the protocols or to the network, because industrial installations are not supposed to be rebuilt to accommodate a discovery layer.

The problem remains open because current mechanisms satisfy some of these properties at the expense of others, and these trade-offs are structural rather than incidental. Centralised service registries can offer expressive queries and low lookup costs, but they concentrate knowledge in a component that must be provisioned, located and kept available, re-introducing the pre-configuration that the paradigm is supposed to suppress.

Distributed alternatives inherited from the Web Services (WS) stack, mainly WS-Discovery in Device Profile for Web Services (DPWS), remove the central registry but keep the message and processing overhead of the Simple Object Access Protocol (SOAP) and eXtensible Markup Language (XML). This overhead can be difficult to justify on devices with processing and communication budgets already dedicated to control tasks. Discovery protocols for constrained networks reduce this overhead but usually restrict matching to identifiers, addresses and type names, leaving the semantic question open. Semantic discovery approaches bring back expressiveness, but typically rely either on an infrastructure component that holds the knowledge base or on a reasoning cost that industrial devices with constraints might not be able to afford.

It is clear that what is missing is therefore not a discovery protocol, but a way of arranging discovery so that these four properties no longer remain mutually exclusive. Chapter 3 examines the existing families of solutions in detail and closes with the specific gap addressed by this work.

1.3 Objectives

Regardless of the conceptual alignment between IEC 61499 and the SOA paradigm, the absence of a standardized interface for automatic service search and for remote calls through messaging protocols still contributes to a complex pipeline design process and to deployment time, keeping the standard misaligned from the modern plug-and-produce concept. The main objective of this work is to investigate how this misalignment can be solved, and to propose a discovery mechanism that closes it.

From this purpose, the work sets out the following objectives:

1. To design a service discovery mechanism that is fully distributed ensuring true plug and produce capabilities under a SOA promise.
2. To keep the mechanism within the reach of constrained devices, both in the traffic it places on the network and in the computation it demands from the participants.
3. To support matching on what a service can actually do, expressed through structured semantic descriptions, rather than on a purely syntactic match over identifiers.
4. To build the mechanism on standardized protocols that can be deployed on existing industrial network infrastructure without protocol extensions, and to demonstrate that its result can be consumed natively by a standard execution environment.

Meeting these objectives requires clear evidences that it is compliant with all the objectives set by this section, so the work also includes the construction of a measurement testbed and a comparison against the discovery stack currently dominant in industrial SOA, under identical conditions.

Finally, the objectives above translate into two research questions:

Research Question 1:

"How can dynamic and semantic resource discovery be achieved at runtime in a distributed CPPS while meeting the limitations of constrained devices and at what cost?"

Research Question 2:

"How may semantic discovery results be used to automate resource binding within an industry standard environment, such as an IEC 61499-compliant runtime?"

1.4 Contributions

Conversely with the objectives set, the original contributions of this dissertation, derived from the architecture proposal, validation and results, are the following:

1. **A two-phase distributed discovery architecture** that performs both syntactic and semantic matching without any registry infrastructure. The first phase encodes the service category in

the Multicast Domain Name System (mDNS) service type and carries a compact semantic summary in the Text Record (TXT), so that the coarse cut happens in the network rather than in the consumer; the second phase retrieves and evaluates a full semantic description over the Constrained Application Protocol (CoAP) only for the candidates that survive the first.

2. **A realisation of the discovery result as an IEC 61499 binding**, in which the endpoint and the interface descriptor produced by a discovery are consumed the requester in an unmodified compliant runtime, proving that semantic discovery and binding can be perfomed during the runtime.

3. **A full prototype description and implementation** which draws the architecture implementation and highlights the design constraints.

4. **A measurement testbed and the architecture's benchmark** for the developed discovery mechanisms in which every reported quantity is a property of the architecture rather than the platform, including the instrumentation needed, the characterization of computational consumption and network overhead analysis.

5. **A quantitative comparison against the discovery stack currently dominant in industrial SOA**, conducted on the same testbed, with the same device population and the same semantic query on both sides, together with a descriptive cost model.

1.5 Document Structure

This document also includes, beyond the introduction, relevant background knowledge to guide the reader towards the problem, its scope and the proposed methodology to address it. In Chapter 2, the relevant theoretical background is presented and sustained with fundamental concepts, standards and architectures. Chapter 3 introduces the state of the art and previous work in the literature along with a discussion of the proposed solutions. Chapter 4 presents the proposed architecture, a two-phase discovery mechanism, together with the rationale and the requirements that shape it. Chapter 5 describes its implementation and the design decisions that only became apparent once there was code to run. Chapter 6 defines what the validation covers, how each quantity is measured and the testbed that follows from both, and Chapter 7 presents the measured results. Chapter 8 interprets those results, uses them to answer the research questions, and sets out the limitations of the method, of the implementation and of the design. Finally, Chapter 9 concludes the work, states the contributions and proposes future developments.

Chapter 2

Background knowledge

There is a broad consensus that, in recent years, the industrial automation field has increasingly implemented concepts and paradigms from other technological domains, particularly from software engineering and information technology. This evolution has been justified and supported by the increasing computational demand of the production systems and the rapid shifting of consumption and production patterns. These new environments require architectures capable of agile adaptation, while operating with minimal human intervention.

Concepts such as modularity, abstraction and distribution have become emergent in the context of CPPS. Common concepts between its definition and various distributed systems paradigms contribute to the convergence of different concepts to achieve the desired modularity in Industry 4.0.

This chapter introduces theoretical knowledge that highlights the resemblance between each core concept contained in this dissertation. Moreover, the key principles of each architecture and specification are exploited to underscore the alignment between them.

2.1 The Industry 4.0 and Cyber-Physical Production Systems

Historically, the industrial landscape and its technological maturity categorized evolutionary milestones. Shifts in the relationship between the consumer trends and manufacturing methodologies highlights the historical boundary between each industrial epoch. These drastic changes in the way the manufacture industries implement different production paradigms started with the adoption of steam machines and mechanization of manual tasks, by the end of 18th century.

Furthermore, in the 20th century, the electricity revolution paved the way for modern production lines. Consume goods started to be produced in mass, with well-defined division on each production labor [45]. Then, the third evolution stage emerged in the second half of the 20th century and was characterized by the rapid advance of electronics and informatics systems as an enabler for the use of embedded systems in industrial context [50]. During this new era, the shop floor machines became capable of executing choreographed algorithms and was also re-shaped

with the adoption of robotics. The first intelligent and automated production lines emerged with the facility of Programmable Logic Controllers (PLCs).

The fourth and most recent industrial revolution, also known as Industry 4.0, is not rooted from the organic manufacturing evolution, but deeply associated to a political strategy of the German government. The "High-Tech Strategy 2020 Action Plan", presented in 2011, resulted in the constitution of a working group and, subsequently, the definition of initial recommendations for the Industry 4.0 [46].

The same document emphasizes that embedded edge devices are becoming even more powerful and network enabled, breaking the boundary between the physical and virtual worlds. Analytically, this advancement paves the way for the definition of Internet of Things (IoT), which is strictly related to CPS spaces. To keep pace with this revolutionary shift in how system components interact to each other through internet services, these two concepts were merged to realize the new industrial epoch.

Integration between physical and virtual processes is strongly discussed by Lee, Edward [31], who defines CPS as the core of this alliance. Besides its strong relation with embedded systems, where processes designed for a specific purpose are confined into a closed loop controller, the CPS concept expands this boundary to an unlimited number of components, virtually linked through network communication. Such coupling enables physical processes to coordinate and control each other virtually in a distributed context. In other words, physical control systems are digitalized and extended to a virtual space, where they can communicate, interact and expand.

This definition strongly contributes to a natural inheritance: manufacture systems realized the CPS mostly using the multi agent-based model, in which every component must be autonomous, decision oriented and distributed [36]. Such approach enables decentralized control and online coordination at the core of CPPS. This was, again, only achievable due to the massive evolution of the IoT paradigm, supported by the large adoption of wireless communication in manufacturing components. Lastly, the conceptualization of CPPS arguably provided a steady ground for the adoption of modern paradigms within the manufacture industry.

2.2 IEC 61499 Standard

The distributed and autonomous behaviour expected from CPPS is, at the shop-floor level, constrained by the way control applications are engineered. The industrial control standard IEC 61131-3 is based on a cyclic scan executed by a single controller, where the application is statically bound to the hardware during engineering and cannot be redistributed without redeployment. This design conflicts with the flexibility and decentralised coordination required by CPPS, motivating an execution model in which control logic is organised as portable and relocatable units instead of a monolithic application. The IEC 61499 standard [26] was designed to provide native support for distributed systems reconfiguration and application portability. It enables algorithm execution into isolated application components called Function Blocks (FBs), eliminating the use of global variables and shared environments. Since the IEC 61499 follows the event-driven pattern design,

the blocks of code (and its internal state machine) are executed whenever the resource detects an input event, which is directly associated with an input data. After the algorithm finishes its execution, the FB changes its state and triggers an output event, which may start the execution of the next block. Figure 2.1 shows that different FBs can reside across different resources, highlighting the main architectural advantage of IEC 61499 towards systems distribution.

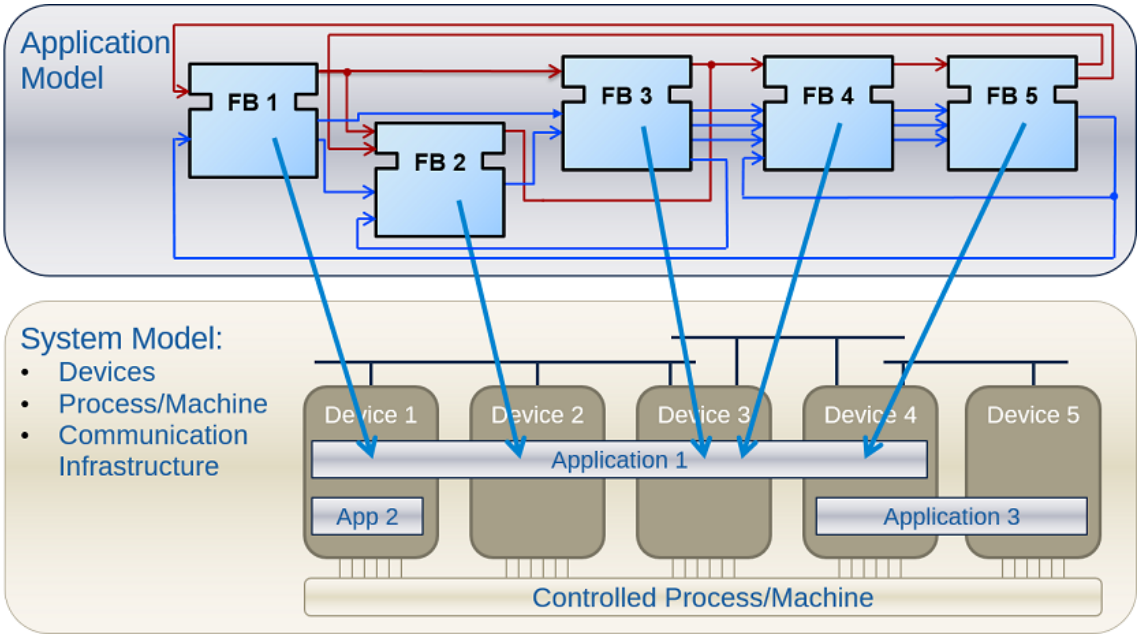


Figure 2.1: IEC 61499 application deployment and distribution. From [51].

The interconnection between FB units is referred to as a Function Block Network. Within this architecture, Composite Function Blocks (CFBs) act as specialized containers that encapsulate an internal network of FBs. Furthermore, Service Interface Function Blocks (SIFBs) provide the necessary interface between the control logic and external resources, such as hardware components.

In the IEC 61499 standard, FBs are executed within Resources, that are managed by a runtime environment. The Resource Manager, which is a component contained within this runtime, manages the FBs execution and it also uses standardized management commands to coordinate the instantiation, life-cycle and migration of these blocks.

There are a few open-source runtime environments capable of executing IEC 61499 applications, including the widely known 4DIAC FORTE, which is a C++ Runtime Environment (RTE) from Eclipse ecosystem [19]. Another suitable open-source alternative is the Dynamic Intelligent Architecture for Software and Modular Reconfiguration (DINASORE) framework [42]. It stands out from other runtime frameworks since it is fully developed in Python programming language, enabling the easy implementation of Machine Learning algorithms and an easier way of programming each component of the pipeline. It is compliant to the IEC 61499 and allows for system reconfiguration also using the open source Integrated Development Environment (IDE) 4DIAC [20].

The standard therefore establishes how control logic is partitioned, deployed and executed across the devices of a distributed application, but deliberately leaves unspecified how those devices expose their capabilities to, and locate, the remaining participants of the system. This complementary concern is addressed by the architectural paradigm discussed in the following sections.

2.3 Service Oriented Architecture

SOA is a software design pattern and architectural paradigm that decomposes complex processes into smaller and discrete software blocks called services [38]. These software components can be individually developed and deployed, also implementing a universal and standardized interface for remote invocation. Unlike traditional monolithic systems, SOA enhances systems distribution connecting these components through messaging, ensuring that the internal logic of a service remains encapsulated and platform independent. Briefly, SOA is not a specific product or technology, but a conceptual framework designed to improve interoperability between systems. [57].

The concept of SOA was originally introduced in the Information Technology (IT) context to overcome compatibility issues introduced by heterogeneous informatics systems. The absence of standardized Web protocols led to the first implementations of SOA with the consequent evolution to the enterprise and business processes [7, 57]. Large applications break its software into modules, uncoupling each functionality to make them independent from each other. Subsequently, different clients can use service interface to interact with the fragment of the original application, independently of its location and underlying platform. While this architectural design ensures the reusability of software, it distributes the capabilities of a target application to an unlimited number of clients.

It is also important to notice that the definition does not limit the resource location to a remote node. Figure 2.2 shows that services consumers and providers can reside both in the same local environment. Since the communication layer may abstract the physical network implementation, the intercommunication of services must follow the same logic as remote service calls.

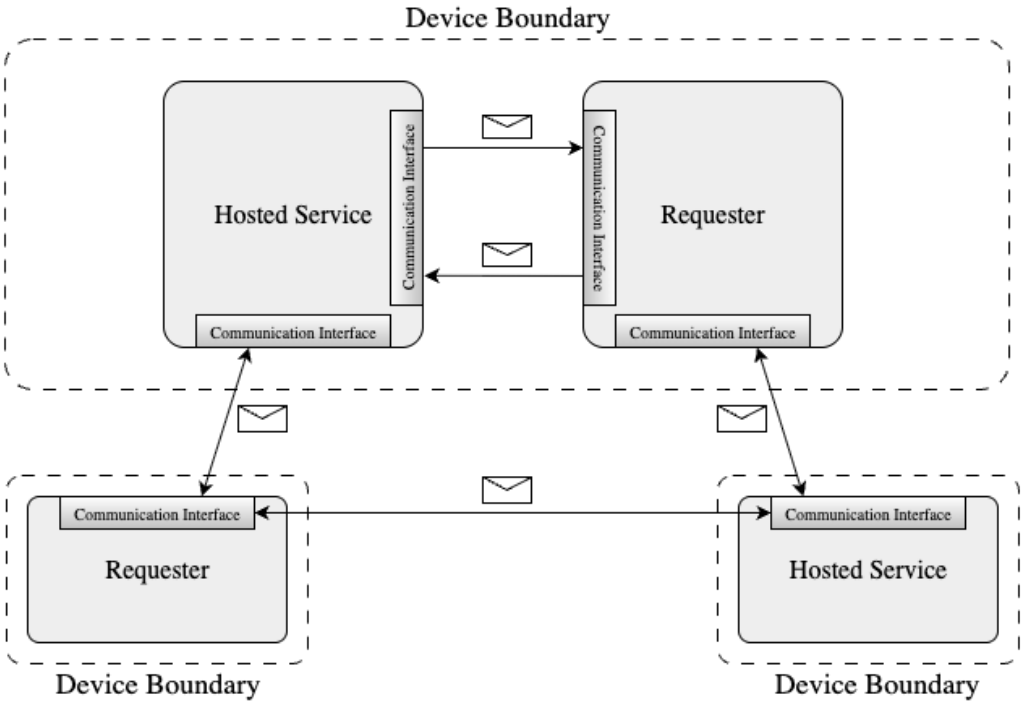


Figure 2.2: Representation of service messaging at different locations.

2.3.1 Key Principles of SOA

In a brief statement, the authors Papazoglou and Heuvel [40] state that "The SOA as a design philosophy is independent of any specific technology". Similarly, Jammes and Smit [28], defined SOA as "(...) a set of architectural tenets for building autonomous yet interoperable systems.". In order to promote such philosophy, a system or application that implements the SOA paradigm to its core must consider several architectural principles. Some of them are:

- **Loose Coupling**: The provided service must be independent from hardware and other system components. In other words, software blocks must implement their functionality with no dependency from each other, preventing the system to be recompiled.
- **Reusability**: Services are designed to be agnostic and context independent. A service must be generically implemented to be callable across different applications.
- **Abstraction**: The service functionality must be implicit and protected, hiding its implementation complexity. An access interface must be provided in order to call it.
- **Discoverability**: Service components must be discoverable by other participants. The metadata, i.e., the service contract, must be exposed and agreed between the parts. The traditional definition of the discovery architecture in SOA considers a centralized service repository, where each available service registers its service contract. Furthermore, different components connected to the system may ask the service repository for available services.

Real-world implementation of the SOA paradigm requires a standardized execution mechanism that implements its principles and technical specifications. In that sense, WS and RESTful applications emerged to standardize and integrate the different layers of the SOA paradigm, supported with the adoption of open protocols.

2.3.2 Synchronous and Asynchronous Interaction Models

Beyond the architectural principles described above, the way in which services exchange information is a defining characteristic of any SOA realization. Two interaction styles are typically considered: synchronous and asynchronous communication. In synchronous interactions, the requester is blocked until the service provider returns a response, which is well-suited for tightly coupled control loops with predictable timing, but introduces undesirable dependencies between the lifecycles of requester and provider [40]. Asynchronous interactions, in contrast, decouple the request from the response in time, typically through message queues, publish/subscribe channels or event-driven mechanisms, which improves resilience to transient faults and supports one-to-many communication patterns [11, 28].

In the context of industrial automation, this distinction is particularly relevant. Control loops with real-time constraints tend to favor synchronous, low-latency interactions, while monitoring, supervisory and analytics functions are naturally aligned with asynchronous, event-based communication. The IEC 61499 execution model, described in Section 2.2, inherently aligns with asynchronous semantics by virtue of its event-driven nature, although its mapping onto network-level service interactions remains an open design choice [13, 14].

2.4 Legacy Web Services and DPWS

While SOA defines the architectural specifications, the WS protocol stack carries out each requirement. The WS-* specification is widely discussed and considered the preferred implementation vehicle for SOA paradigm [28].

The realization of the WS specifications pivots on XML. Its presence through the WS-* family protocols enhances the standardization on how layers communicate. It also enables semantic interoperability between different components and services of a system, ensuring the communication is unambiguous and human-readable. The adoption of this unified description language facilitates the integration of interrelated technologies to build a web service [53].

XML serialization and data format is not sufficient to bridge the communication through all the components of the WS-* family. For that reason, SOAP was introduced as a protocol capable of constructing and exchanging information between nodes and the underlying structure of a Web Service. Previous versions of the protocol definition introduced two ambiguous interpretations of its acronym: Service Oriented Architecture Protocol, in general meaning the atomic SOAP message for general purpose, and Simple Object Access Protocol, where the SOAP message represents a remote method invocation [53].

Since each service provider may implicitly implement its functionality, a normalized interface should describe the remote functionality to other participants. The Web Services Description Language (WSDL) describes how a client can interact with a remote service, containing also data types of the expected message content upon a request [39]. This mechanism abstracts the remote algorithm implementation exposing only the interface – defined by the WSDL document – to the network.

2.4.1 Web Services for Industrial Embedded Devices

The implementation of Web Services in industrial environments became a widely discussed topic in the early 2000s. The great potential of SOA paradigm to enhance systems interoperability, and its rapid adoption in the IT field, drove interest in the manufacturing community. The Service Infrastructure for Real-time Embedded Networked Applications (SIRENA) has become a pioneer in the research of SOA for manufacturing, proving the capability of Web Services to be a suitable implementation of SOA for edge devices [28]. The main motivation for the research was the rapid changes on consumption patterns. Such constantly changing requirements, with the introduction of new technologies at the shop floor, demanded a new level of flexibility and agility without compromising the availability of the system.

Within this context, The SIRENA project realized the SOA paradigm through the implementation of DPWS for embedded and real-time devices. In fact, the project contributed to the consolidation of the DPWS stack as a suitable implementation of the SOA paradigm for embedded edge devices. The DPWS stack allows not only for the design of autonomous service discovery, through WS-Discovery protocol, but also the standardization of many layers of the Web Services stack, like messaging (SOAP), addressing (WS-Addressing) and service description (WSDL) using XML scheme [52]. A Web Service profile can be tailored to the application given its requirements, which means there is no explicit guideline to implement the Web Service stack on the target device (nor its client) [56]. An example of DPWS composition is given in 2.3, where the target device is capable of discovering other devices on the network, via User Datagram Protocol (UDP) multicast probing, expose its service description/interface through WSDL and exchange messages using SOAP protocol. Note that the DPWS stack does not necessarily reflect the physical architecture related to the Open Systems Interconnection (OSI) model, but a logical layered abstraction of each of its components.

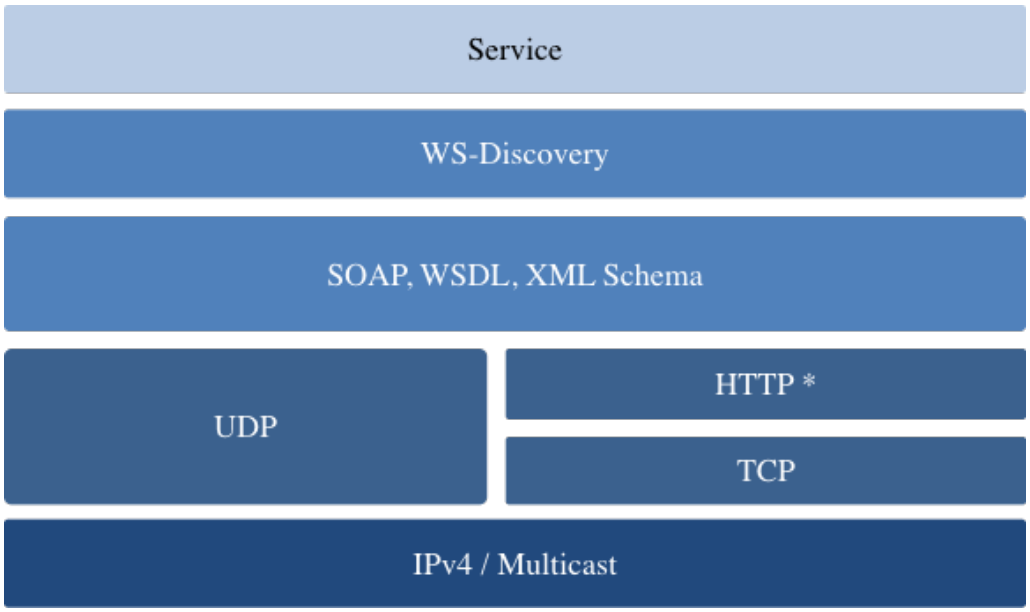


Figure 2.3: Simplified layered version of the DPWS protocol stack.

2.5 RESTful Web Services and Constrained Protocols

The Web Services protocols, discussed in Section 2.4, represent what the literature commonly refers to as the *Big Web Services* family [41], characterized by XML-based payloads, SOAP envelopes and a rich set of orthogonal specifications layered on top of HyperText Transfer Protocol (HTTP). While this stack offers a comprehensive coverage of enterprise requirements, its verbosity and processing overhead are, in most of the cases, heavyweight for resource-constrained devices and time sensitive applications.

A complementary architectural style, formalized by Fielding [18] in his doctoral dissertation, proposes a resource-centric view of distributed systems. In the Representational State Transfer (REST) style, every entity of interest is modeled as a uniquely addressable resource, represented as Uniform Resource Identifiers (URIs), and manipulated through a fixed and uniform set of operations, while the interaction itself remains stateless [18, 41]. This minimalist contract reduces the coupling between client and server and removes the need for an explicit description language at the transport level.

The most widespread realization of REST relies directly on HTTP, where the protocol verbs (*GET*, *POST*, *PUT*, *DELETE*) are reused as the uniform interface and the resource representations are typically encoded in JavaScript Object Notation (JSON). This combination is often referred to as *RESTful Web Services* and has progressively displaced SOAP in scenarios where lightness, ease of integration and developer accessibility outweigh the formal guarantees of the WS-* stack [41].

2.5.1 CoAP Ecosystem

To overcome some assumptions imposed by the RESTful HTTP realization, such as reliable Transmission Control Protocol (TCP) transport and persistent connections, and to enable low-powered devices to benefit from the REST interaction model, the Internet Engineering Task Force (IETF) introduced CoAP [48], which runs natively over UDP, employs a compact binary header and supports the same verbs as HTTP.

Several CoAP extensions enhance the basic request–response communication model to better support typical requirements of IoT systems. For example, the Observe option [23] enables a publish/subscribe interaction on top of the request/response model. Another important extension is the Block-wise transfers [5], which allows the protocol to split payloads larger than a single UDP datagram into ordered fragments, while preserving the stateless nature of each individual exchange. Finally, the Constrained RESTful Environments (CoRE) Link Format [47], in the context of resource directories, provides a compact textual encoding for advertising the resources hosted by a CoAP endpoint, together with attributes such as resource type (*rt*), interface (*if*) and content format (*ct*).

2.6 Service Discovery Mechanisms

Service discovery is the process by which a service consumer locates a suitable service provider in a distributed system. The choice of discovery mechanism has a deep impact on the system's modularity, scalability and resilience, and largely defines the extent to which true plug-and-produce capabilities can be achieved [1, 6]. Discovery mechanisms can be broadly classified along two orthogonal axes: the architectural arrangement of the discovery process (centralized/distributed) and the type of matching process (syntactic/semantic).

2.6.1 Centralized Registry-based Discovery Architecture

Early SOA implementations relied on centralized directories such as the Universal Description, Discovery and Integration (UDDI) registry, in which providers publish service contracts and consumers query the registry to retrieve them [40].

The Open Platform Communications Unified Architecture (OPC UA) [24], a standardized middleware frequently used in Industry 4.0 context, implements this centralized discovery mechanism through the Global Discovery Server (GDS) component. In this model, every endpoint capable of providing one or more services registers its application URI, server certificate and the set of exposed objects in the GDS, which additionally assumes the role of a Certificate Authority for the secure channels established during each subsequent session [25]. To discover these endpoints and their respective services, a client must connect to the GDS and execute a search query via Remote Procedure Call (RPC). The GDS then returns a list of available server endpoints.

An equivalent realization can be observed within the WS-* family through the managed mode of the WS-Discovery specification: a discovery proxy concentrates announcements that would

otherwise be exchanged via multicast and answers unicast probes on behalf of the registered devices [54, 56]. The discovery proxy plays, in essence, the same architectural role as the OPC UA GDS or the UDDI registry.

The same design rationale has been transposed to the CoAP ecosystem through the CoRE Resource Directory (RD), recently consolidated as Request for Comments (RFC) 9176 [3]. The RD entity acts as a proxy that aggregates all the available nodes that are sleeping behind a network segment, or otherwise unable to respond multicast queries. Devices register their abilities using the CoRE Link Format encoding [47]. Clients can further perform lookup operations against the RD also using CoRE standard request format over its exposed interface. Optionally, clients can filter attributes such as resource type (*rt*), interface (*if*) or content format (*ct*) [6, 47].

Centralized approaches are conceptually simple and efficient in steady state, but introduce single points of failure and require explicit configuration of the registry endpoint, which conflicts with the zero-configuration philosophy expected from modern CPPS.

2.6.2 Distributed Discovery Architecture

Decentralized alternatives leverage Internet Protocol (IP) multicast and overlayed networks to disseminate discovery messages without relying on a fixed directory.

The most widely deployed instance of this paradigm is the ad-hoc mode of WS-Discovery, in which devices announce their presence and answer probe requests via SOAP-over-UDP messages sent to a well-known multicast group, without any intermediate proxy [54, 56]. It has been the dominant choice in industrial SOA prototypes for over a decade due to its tight integration with the remaining WS-* stack and its native support for type-based and scope-based filtering.

In the IoT domain, the dominant counterparts build on the Domain Name System (DNS): mDNS [9] and DNS-Based Service Discovery (DNS-SD) [8], which together implement a fully decentralized name resolution and service advertisement mechanism over the local link. In this combination, each node acts as its own authoritative name server for a portion of the *.local.* namespace, answering multicast queries for the records it owns and probing the network to detect name conflicts before claiming a unique identifier [9]. Service instances are advertised through Pointer Record (PTR), Service Record (SRV), and TXT records, where the TXT record carries arbitrary key/value metadata describing capabilities, versions or vendor-specific attributes within the 255-byte-per-string limit recommended by the standard [8]. The combination of these two RFCs has effectively become the de facto zero-configuration substrate of consumer and industrial IoT, being adopted by great companies to realize domestic IoT plug-and-play protocols, such as Apple Bonjour and Google Cast [4, 21].

CoAP also supports multicast-based discovery, in which clients send GET requests to the "all-CoAP-nodes" multicast address against the /.well-known/core URI defined by the CoRE Link Format [47]. Each responding server returns a link-format payload describing the resources it hosts, optionally filtered by attributes such as resource type (rt) or interface (if). This approach preserves the zero-configuration nature of mDNS while staying within the UDP/CoAP footprint

expected from heavily constrained devices, and it has been explored as the substrate of several semantic discovery extensions

All the above mechanisms are, in their canonical form, syntactic: they match services based on identifiers, types or attribute keywords. Semantic discovery extends this matching process with formal descriptions of service capabilities, typically expressed using ontologies and Semantic Web technologies such as Resource Description Framework (RDF), Web Ontology Language (OWL), or OWL for Services (OWL-S) [55]. This allows consumers to query for services in terms of what they do rather than how they are named, which is particularly valuable in heterogeneous industrial environments where naming conventions and vendor-specific terminology vary widely.

Chapter 3

State of the Art

The increasing adoption of distributed automation systems has intensified the need for flexible, interoperable, and scalable execution models. Within this context, the IEC 61499 standard has emerged as a suitable framework for distributed control, particularly when combined with SOA principles. This chapter reviews the state of the art in the integration of SOA paradigms with CPPS, focusing on service-oriented execution environments and autonomous service discovery mechanisms.

Furthermore, this chapter examines IT-driven applications that have successfully addressed gaps similar to those identified in this dissertation, particularly regarding Plug-and-Play and IoT integration challenges. These contributions show, as in the past, a very promising path for a solution that can be applied to an industrial context.

The reviewed works are analyzed in terms of architectural design choices, communication protocols, and runtime support, highlighting existing gaps that motivate the contributions of this dissertation.

3.1 Service-Oriented Architectures in Cyber Production Systems

Candido et al. [7] realised the combination of SOA paradigm and Evolvable Production Systems [33] by proposing a model that can be dynamically deployed over services. The authors emphasize classic automation issues, such as long setup and deployment times and incompatibility between different vendors. The proposed architecture separates devices services into: Built-in Services, which are natively embedded into the physical device and are immediately available after the setup, and User Services, deployed dynamically during the runtime. Moreover, the authors also defined how services are managed and deployed to each physical device as well as the system infrastructure, illustrated in Figure 3.1. The most important components of the proposed architecture includes:

- Service Deployer: A software component, exposed as a network service, responsible for dynamically deploying new services into physical devices;

- Service Repository: A catalog of logical services that can be downloaded;
- Device Explorer: A software component capable of discovering, checking the status, visualizing metadata and testing available devices on the network;
- Service Design: The set of tools used to design and reconfigure the application based on the available devices and services.

The authors also refer that, a logical topology of the deployed application could be obtained combining the metadata of each logical device. This is an important statement considering the scope of this dissertation, since the decoupling and abstraction between different nodes implies that individual devices are not aware of specific services running on its neighbours. Therefore, exploiting local metadata can provide an interesting method for creating an ontological knowledge.

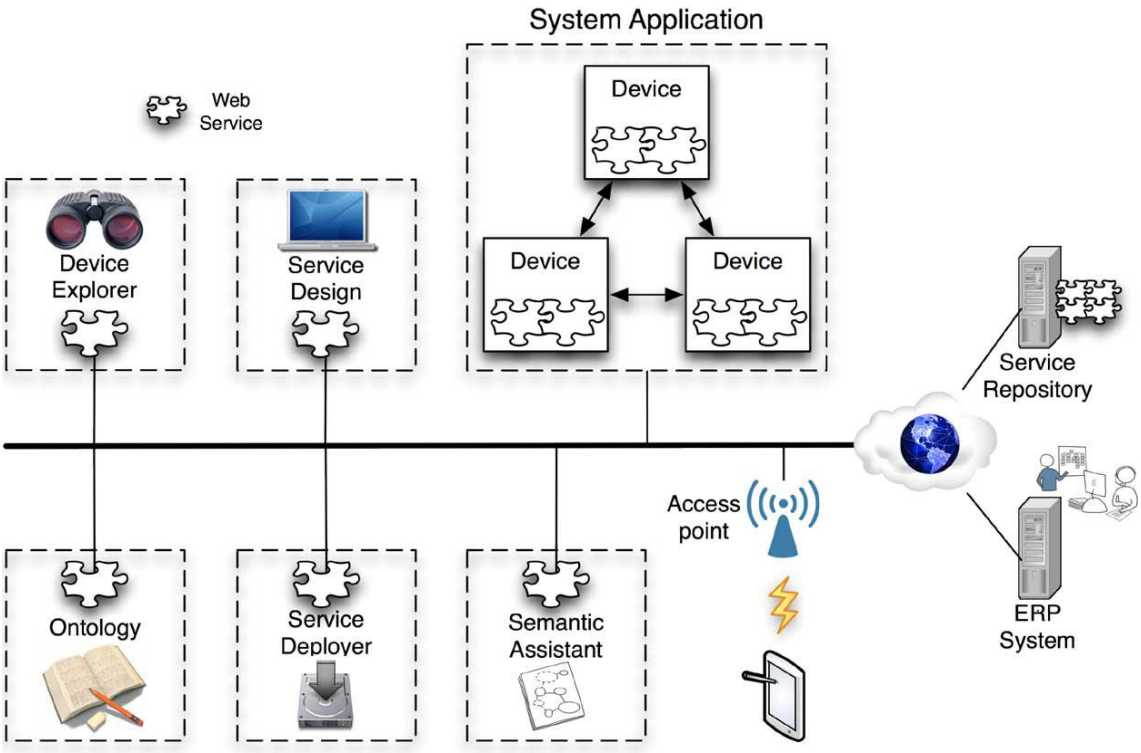


Figure 3.1: Service Oriented infrastructure. From [7].

To validate the developed solution, the authors translated the traditional control code of a packaging production line, developed under the IEC 61131 standard, which defines the programming languages for the cyclic-scan control model that preceded IEC 61499 and remains dominant in industrial practice, into PLCOpen format. In this scenario, a dosing machine is responsible for filling recipients, attached to pallets, with granules. The orchestration between each control component, including a PLC for controlling the pallet indexer and Radio-Frequency Identification system, as well as low-power Input/Output (IO) controllers, was fully executed by service calls. The device and services information were defined using an XML configuration file, which is then

dynamically modified by the Service Deployer component using the WS-Management protocol. For service discovery and invocation, the DPWS stack was employed.

A more systematic investigation of bridging the SOA design to the Cyber Production System environment was discussed by Dai et al. [13], who proposed an external interface to bridge the event driven architecture of IEC 61499 standard to the messaging pattern required by the SOA architecture. This work also introduces a basic structure, illustrated in Figure 3.2, for the flow of messages in a function block pipeline, easing the comprehension that a simple FB constitutes a service requester and, conversely, a service responder. In other words, if an FB triggers an output event, it is requesting a service, which is interpreted as a request for a service by the neighboring block.

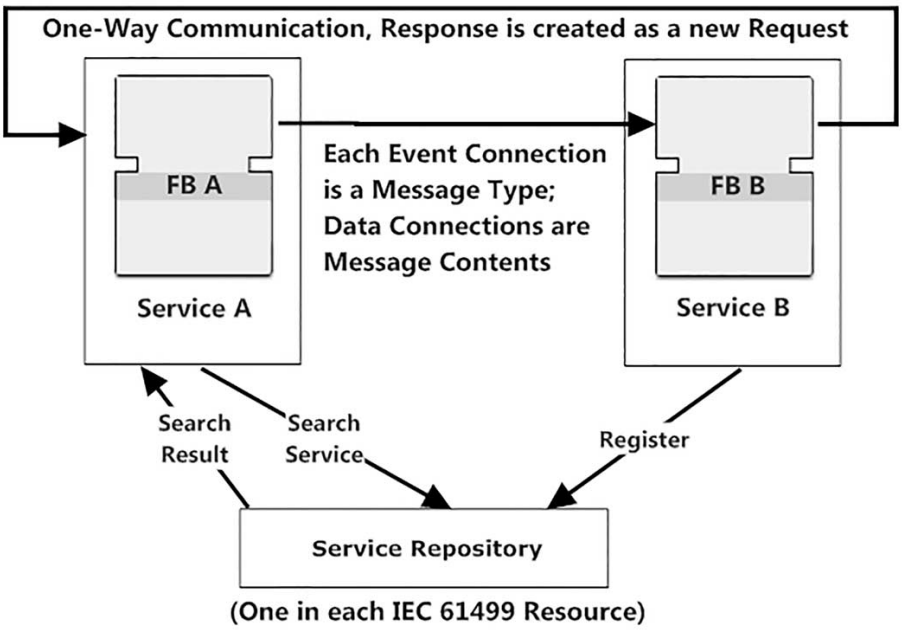


Figure 3.2: Basic SOA architecture for IEC 61499 standard. From [13]

The integration between the SOA paradigm and the IEC 61499 execution environment proposed by the authors was supported by widely adopted and standardized Web Service protocols to address different architectural layers. The communication layer was realized over the TCP/IP stack using SOAP messages, thereby enabling platform-independent communication between each component. The management commands defined in the IEC 61499 standard, such as *CREATE*, *DELETE*, *START* and *STOP* were encapsulated in the body of these messages, facilitating the system reconfiguration at runtime. In parallel, at the discovery level, the authors chose to implement the WS-Discovery protocol. In the scope of the developed research, services discovered during the runtime are added to the service repository maintained by the Resource Manager, which is the central piece responsible for interpreting the management commands from the IEC 61499 IDE and maintaining service contracts.

Although the runtime environment proposed by the authors outperformed in latency the FORTE C++ runtime [19] (due to the delay introduced by the SOAP messaging over TCP/IP stack), the

architecture demonstrated superior flexibility and SOA compliance. Finally, the plug-and-play supported by the WS-Discovery protocol was not deeply discussed by the authors, leaving an important gap regarding one of the SOA specifications: a service must be discoverable.

Partially addressing computational constraints, the model proposed by Demin et al. [16] highlights the use of cloud computing services to overcome computational limitations for time-consuming tasks. The authors defend that migrating such control logic tasks to a cloud environment can ease service upgrades and enhance the system modularity and reusability. For demonstration purposes, a 6-cylinders Pick-and-Place machine was virtually setup on top of the IEC 61499 FB pipeline supported by a distributed control mechanism. The operation of each pneumatic cylinder is controlled by a CFB, which can be internally divided into two simple FBs: the *CylSchd* FB, responsible of planning the execution, and the *CYLACT* FB, responsible for the actuators control.

The system is capable of efficiently scheduling each cylinder's operation, with exhaustive evaluation of all possible combinations. The cloud-based scheduling service also considers multiple optimization criteria, such as minimizing the overall energy consumption. Service requests were implemented using the SOAP protocol transported over HTTP. Furthermore, a Composite Automation Types-block – a conceptual extension from the discontinued nxtStudio environment – was used to encapsulate the service interface enabling the importing of a custom .NET Dynamic-Link Library (DLL) to the system.

Although emphasizing the advantages of cloud-based potential over a regular PLC hardware capabilities, the study did not present a formal evaluation metric that could quantify the performance gain. Additionally, the proposed model introduces a minor architectural contradiction in the scope of distributed systems: the authors favor a centralized dispatcher Composite Automation Types-block over the autonomy of each cylinder controller. Of course this design decision aimed to reduce the network overhead, however, considering that the system operates on a relatively simple data structure, such overhead would likely be minimal.

The developments from Candido et al. and Dai et al. [7, 16], contribute for the general consensus developed on this dissertation: legacy web service protocols, such as SOAP for messaging and XML for service composition and description, are heavy and require unnecessary computational power. Moreover, the joint between the application and network level must be enhanced to support constrained hardware to operate under the SOA requirements.

A later contribution for bridging SOA to CPPS with a lightweight trace is presented by Dai, Huang, and Vyatkin [14], who emphasized the actual transition from the International Society of Automation (ISA)-95 hierarchical architecture to a two-layered distributed organization, separating the industrial automation system into Cloud and Edge computing spaces. The deployment of IEC 61499 compliant FBs is performed in a containerized resource. Each FB instance acts like a microservice, the event and data connections between instances deployed in the same resource are realized via inter-process communication and shared memory. On the other hand, the communication between different resources is separated into synchronous and asynchronous interactions. To reduce the overhead of the communication process, previously developed over the legacy WS* stack, the authors implemented the IEC 61499 management commands as HTTP REST requests.

Although contributing for plug-and-produce paradigm, the matching process adopted from the previous authors rely on the basic syntactic matchmaking, which means they do not take into consideration the specificities required in nowadays production pipelines. Finding a resource that abstracts from functional and non-functional requirements might lead to mismatching of interfaces, capabilities and geographical limitations.

An interesting move to break into semantic space is introduced by Lu and Ju [35], who implemented a 3D printing service under industrial context utilizing the *Status Enterprise* software (compliant with OPC UA specification) to expose service metadata, such as the printer queue length and remaining time. For this purpose, they proposed a new architectural model called Cyber-Physical Manufacturing Systems (CPMS), in which physical assets are encapsulated into object-oriented service modules. This model defines a CPMS service by joining the *Service-Capability* and *ServiceInterface* objects: the first defines the service attributes such as capacity, availability, and material it can process, while the second object defines the service description, input and output types. These services are then discovered using a centralized Universally Unique Identifier (UUID)-based registry and WSDL interfaces, and the capabilities of each service are mediated by reference ontologies (Process Specification Language (PSL) and MAnufacturing's Semantics ONtology (MASON)) to match the requested service with the available physical resource.

Despite the conservativeness, meaning it was developed over some legacy architecture such as the widely adopted OPC UA and WSDL for interface description, the work presented by Lu and Ju [35] serves as a first move that guided this dissertation into the semantic trend on CPPS environments. A concept that is highly coupled with semantic service description is the mechanism behind its discovery, which plays an important role when considering evolvable CPPS and flexibility of the production system. The architectural choices to implement discovery mechanisms are also constrained by the physical and architectural requirements, meaning it demands special attention on this dissertation.

3.1.1 Service Discovery and Composition

The tendency for autonomous service discovery in CPPS environments noted by Candido et al., Dai et al., and Lu et al. [7, 13, 35] supports the need for an effective way of representing services and their specifications, as well as a robust mechanism for supporting discovery and matching capabilities at runtime.

The proof of concept presented in [12] extends the Function Block Service Runtime (FBSRT) developed in [13] with a richer subset of WS-* protocols on top of the DPWS stack (Fig. 3.3). The authors aim to suppress the autonomous discovery limitation under the justification that modern automation and distributed systems still struggles with the growing complexity and increasing number of interconnected devices.

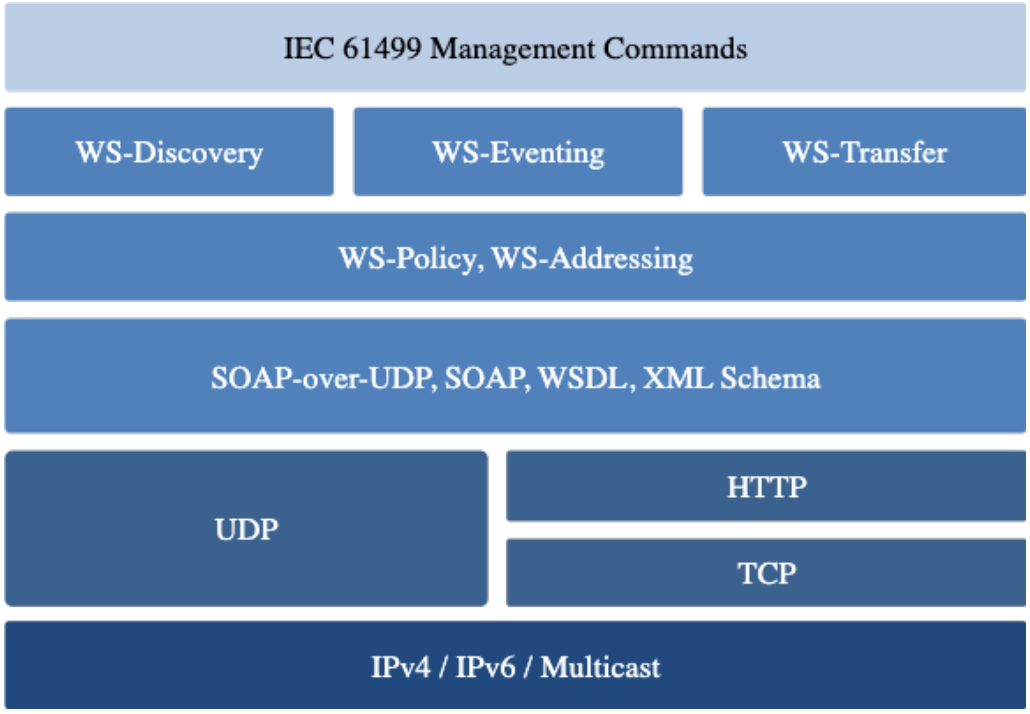


Figure 3.3: Layered structure of the DPWS protocol stack for IEC 61499 management. Adapted from [12].

The validation setup chosen by the authors consists in a building lighting system, where each room has its own FBSRT controller. For every Light-Emitting Diode (LED) ballast in a room, a new service is instantiated and its capabilities are then exposed. One of the tests performed, focused on evaluating the DPWS stack implemented to the model: an initial service search found the target LED ballast, then the mapping of the IEC 61499 to the WS-Transfer protocol to retrieve the service contract and, finally, the adjustment of the target LED's brightness via a service request.

Beyond introducing real plug-and-produce functionality, this work is also important because it implements service requirements within the service metadata, correctly mapping data types and input/output parameters. Such description might be considered a basic semantic approach.

As already mentioned, many technologies and protocols adopted on the IT domain have been transitioned to the industrial context, with the aim of providing more flexible and scalable architectures. Under this scope, Profanter et al. [43], proposed a solution for the discovery of devices and services in a manufacturing plant, based on the OPC UA with Local Discovery Server with Multicast Extension (LDS-ME) mechanism [25]. The authors demonstrated a hierarchical Plug-and-Produce discovery architecture in which a central bus discovers the workstations, and each workstation discovers its own sensors and actuators via multicast probe. Implementing the LDS-ME mechanism was done using the mDNS protocol and the experiments revealed that a device can discover, register and query the OPC UA server in less than a second. Finally, the authors highlighted the fact that different implementations of the mDNS protocol to realize the LDS-ME mechanism can lead to different performance results, with some implementations taking up to 7 seconds to discover a device due to internal language specific library timeouts.

In a more network-driven approach, Ismail and Kastner [27] adapted the concept of the Enterprise Service Bus (a software pattern for integrating different applications) to the industrial context. The result of this adaptation is a distributed Gateway Service Bus architecture for discovering services and enabling vertical integration in a manufacturing plant. Each Gateway Service Bus node implements a basic set of three services: networking, discovery, and management. When discovering a neighboring node, the discovery service looks for the provided capabilities and forwards this information to the management service. The management service then acts as a decision engine, determining which overlays the node should join, and consequently instructs the networking service (running a modified Chimera protocol) to update the routing tables of the cooperative overlay network. By adopting the mDNS protocol for the discovery service, the authors aimed to suppress the overhead of traditional enterprise discovery protocols, such as WS-Discovery. The system is capable of discovering Gateway Service Bus nodes at different subnets by using a mDNS reflector and was validated in a testbed with 11 virtual machines distributed in two different subnets. The results revealed that the system can guarantee the discovery of all the services in less than 3 seconds, with a significantly lower overhead when compared to the XML-based WS-Discovery (300 bytes against 1088 bytes per packet).

The mDNS approach for service discovery in industrial context presented above highlights that IoT protocols are fully capable of solving the same discovery problem out of the IoT scope. The possibility to reflect the mDNS packets through different subnets might enable service discovery within different network contexts, which are commonly deployed at wide enterprise environments. Considering the scope of this dissertation, adopting an application protocol that can be easily deployed into existing network infrastructure, without disrupting operational workflows, would be optimal.

A more recent approach is discussed by Nguyen et al. [37], who proposed an integration of the Web of Things (WoT) paradigm to the discovery of industrial field devices, but also focusing on the semantic self-description of these devices. The implementation of the proposed architecture is based on WoT Thing Description (TD) encoded in JavaScript Object Notation for Linked Data (JSON-LD) file, which describes the capabilities in a metadata format that can be directly advertised to an OPC UA server. Service announcements are performed using the WoT Discovery mechanism via UDP multicast messages and, once received by the OPC UA server, the TD is used to dynamically update the server's address space. Although solving the static information models approach, the manual writing of these descriptions is considered a complex task in systems with a large number of devices, and the current WoT vocabulary still lacks specific terms for industrial data types natively supported by OPC UA. In contrast, the multicast approach to implement a distributed way to update the legacy OPC UA address space, and the adoption of a lighter description file, arouses the attention of this dissertation to study a light and distributed solution for service discovery.

The literature reviewed in this section share a common form. Those that eliminate the registry, like the DPWS lineage, suffer from larger message sizes since every exchange includes a SOAP envelope over XML on a network that also handles control traffic. Approaches based on mDNS

keep traffic low by reducing matching to a service type, leaving the consumer unable to determine if a device of the right type can perform the task. Those that regain expressiveness, such as the OPC UA information model, introduce a component that must be found before it can be queried, or a browsing cost that devices with limitations cannot handle. Each approach has its strengths and answers a different question. However, none address how a consumer decides which devices, without infrastructure and without extensive exchanges, are worth interacting with.

3.1.2 Semantic Description and Matchmaking

Integrating heterogeneous systems has turned more complex over the years. Production lines have become even more capable of dealing with dynamic SOA composition and specificities of new edge devices. An iteration to solve this heterogeneous problem is proposed by Jirkovsky et al. [29] who aimed to optimize the stop-problem of a hydroelectric power plant, identifying accurately the optimal moment to restart the turbines for auto-cleaning the vanes. For this purpose, 38 sensors were installed across the power plant to acquire variables such as water fall and power factor. Reading the sensors data is performed in a synchronous manner at every 5 seconds, generating 656,640 samples per day. Each sensor runs an OPC UA server which incorporates semantic metadata into the server's address space. This metadata is modeled using the Turtle language and defines the device based on the ontology also developed by the authors: the Cyber-Physical System Ontology for Components Integration. Once discovered by the Semantic Big Data Historian – the central system who also implements the OPC UA client – the sensor metadata is extracted and processed.

The dynamism of the proposed solution also lays on the way the system manages the devices after first connection: if the resource uses a previously known concept, its data is immediately sent for storage or processing. However, if the device presents an unknown concept, but its metadata indicates that it derives from a super-concept already existing in the ontology, the system automatically extends its own Cyber-Physical System Ontology for Components Integration ontology to incorporate the new definition. This mechanism ensures that the system understands the meaning of the new data without human intervention, making it available to the analytical layer through flexible SPARQL Protocol and RDF Query Language (SPARQL) queries.

Directly contesting the implementation of distributed semantic metadata, Lober et al. [34] argue that browsing through the OPC UA information model to extract semantic information of a target device is not efficient, especially for wide environments. For this reason, the authors implemented an Ontology Management System that acts as a centralized registry where the equipments are already modeled within the ontology and ready to be queried via SPARQL. The authors demonstrated this concept using a logistics use case in which, instead of requesting a predefined service by its syntactic name, the digital twin of the stored product searches the Ontology Management System for any available transport capable of carrying a pallet with a minimum weight capacity, given in kg, and a minimum speed, in m/s. Consequently, the ontology analyzed these physical requirements and returns appropriate endpoints for Automated Guided Vehicles and forklifts.

The argument defended by Lober et al. is totally comprehensive, since OPC UA is computationally expensive for resource-constrained devices. Taking into consideration environments with thousands of devices, the metadata extraction would likely be unfeasible. However, adopting a centralized way for retrieving service semantics forgoes the benefits of plug-and-produce in the sense it would still require pre-configuration. Additionally, centralized approaches might lead to a single point of failure, compromising the entire plug-and-produce promise.

Shifting the focus to the control logic, Dai et al. [15] address the flexibility and semantic problem by developing a knowledge-based and microservice-based model that enables semantic plug-and-play for industrial edge controller. The authors employed the same base architecture stated in a further work, already described in Section 3.1 [14], but the focus of this work spots the employment of the Monitor-Analyse-Plan-Execute-Knowledge industrial software agent, enriched with semantic capabilities. Developed over the IEC 61499 standard, the plug-and-play capability was employed and validated in an assembly line case study, where the routing plan of an Automated Guided Vehicle was dynamically reconfigured across different workstations. When a change in the manufacturing process is required, the agent queries the knowledge base, through Semantic Query-Enhanced Web Rule Language (SQWRL), removes obsolete FB bindings, and creates new event and data connections on the fly.

The various semantic approaches can be broadly divided into two categories, and this distinction is fundamental. When the knowledge base is centralized in a single component, as in the Ontology Management System, queries become more expressive while reducing the computational burden on individual devices. However, this benefit comes at the cost of provisioning, locating, and maintaining a central authority , whic is precisely the kind of pre-configuration that the plug-and-produce approach aims to avoid.

In contrast, distributing semantic descriptions across the devices themselves preserves the plug-and-produce paradigm, but shifts the reasoning burden to resource-constrained devices. It also requires users to retrieve descriptions from every potential candidate before any can be ruled out, increasing communication overhead.

Ultimately, the key question is where semantic knowledge should reside and who should pay the cost of interpreting it. Rather than choosing one approach over the other, a mechanism that combines elements of both would be the optimal model.

3.2 Relevant Contributions from IT

Outside the CPPS ecosystem, the IoT community has explored analogous semantic discovery mechanisms at larger scale. Although conceptually powerful, semantic matching approaches are difficult to implement on resource-constrained devices due to the overhead of the entire reasoning process. Building on this baseline, several authors have proposed semantic extensions for discovery and service composition using emergent technologies from IT field.

The emerging CoAP protocol is studied by Yachir et al. [55] and Ruta et al. [44]. The first identified that semantic attributes can be built on top of the CoRE Link Format, which means that

devices can discover services by composing semantic filters directly in the CoAP URI. The request is then processed by the CoRE RD using SPARQL queries and returns the appropriate endpoint if the requirements are met, including service and device Quality of Service (QoS). The registration to the RD uses the standard CoAP attributes combined with extended semantic fields to explicitly identify the QoS, location, and semantic ontology reference. Ruta et al. [44] in contrast, embedded the semantic description within the CoAP payload using Description Logics (DL). The authors identified that exact request-to-resource matches are extremely rare in highly heterogeneous and dynamic IoT environments. To address this problem, they proposed a more decentralized alternative and also introduced the concepts of Concept Abduction and Concept Covering which, together, enabled the gateway of the system to calculate the remaining requirements (the Hypothesis) that are not fully satisfied by the local sensors. Then, a collaborative algorithm expands the search for uncovered part of the request to neighboring gateways, incrementing a hop variable. At the end, the system is capable of synthesizing the final response by aggregating the distributed resources that collectively fulfill the original request, successfully supporting approximate matches.

A similar approach for the semantic description developed by Yachir et al. [55] is developed on top of the DNS protocol. Fernandez et al. [17] proposed a compression mechanism that encodes device semantic metadata into compact binary identifiers. Sensor type, geographical and logic location and their data context are encoded in Base32 to form a valid self-certifying DNS name. The authors also utilized the DNS-SD mechanism, where a client queries for PTR records to find services that match that encoded semantic name. Additionally, they propose using DNS TXT records to store and publish the sensor data itself, taking advantage of DNS caching.

This architecture demonstrates that mDNS, which derives from DNS-SD, can incorporate a significant pre-filtering layer without requiring additional infrastructure. Implementing such mechanism to exclude resource candidates that do not satisfy the functional (and additionally non-functional) requirements would optimize the further metadata extraction process. Moreover, this contribution can be directly correlated to the one developed by Lober et al. [34], at Subsection 3.1.2, who criticized cyclical metadata extraction. However, this approach requires consensus on the encoding scheme and does not address how the underlying ontology might evolve over time. Implementing a multilayer discovery mechanism is a promising path that can be explored to address semantic discovery for computationally constrained devices.

The literature has previously anticipated the integration of mDNS and CoAP to harness the benefits of both technologies. For instance, Cirani et al. [10] developed a two-layered architecture for service discovery, merging the Zero-Configuration (ZeroConf) framework - which heavily relies on mDNS - and global Peer-to-Peer (P2P) networks. An edge gateway manages the traffic between the local environment and external overlay networks. Once the edge gateway detects a mDNS packet announcing the presence of a new device, it extracts the resource capabilities through a CoAP GET request targeted at the standard interface */.well-known/core*, subsequently storing the information in a Local Resource Directory (LRD).

To overcome the single point of failure problem and ensure global scalability, the authors divided the interconnection of the edge gateways into two distinct overlays: the Distributed Location

Service (DLS), which acts as a Distributed Hash Table (DHT) for deterministic URI resolution, and the Distributed Geographic Table (DGT), which routes the requests based on the geographical coordinates of the gateways. Once discovered locally, the gateway publishes its LRD into the DLS, making the services globally reachable. Consequently, the edge gateway functions as an HTTP-to-CoAP proxy to bridge the external clients with the constrained devices.

Finally, since both technologies are architecturally compliant and lightweight against legacy protocols currently diffused in the CPPS context, the development of a semantic architecture that merges both technologies may take advantage of the scalability and consolidation of mDNS and the REST lightness of CoAP. By doing so, the resulting architecture would benefit from a distributed service discovery support and semantic transfer capabilities. Also, beyond the literature contribution, the cooperation between CoAP and mDNS was explored and documented in a standard draft maintained by IETF [49]. Even though the draft was never published, it contains valuable information about the integration of both technologies to achieve efficient resource and service discovery.

3.3 Conclusion

The literature review presented in this chapter highlights the efforts between different authors and their clear trend towards the adoption of a modular and SOA compliant architectures. In the context of CPPS, the analyzed works, in the logical way they are presented, emphasizes the relevance of SOA paradigm and how it can potentially bridge resources through a standardized discovering mechanism.

By diving into the literature, it becomes noticeable that there is no architecture that simultaneously provides:

- **Decentralized Discovery**: which mitigates the single point of failure and enhances the plug-and-produce with no need of human intervention. Although having known limitations, such as increasing network overhead, it is promising in the way it naturally introduces flexibility and modularity, characteristics that are SOA compliant. Despite the advancements, qualitative comparisons between most recent approaches for service discovery and plug-and-produce mechanisms are still absent;

- **Lightweight Operation**: ideal for computationally-constrained devices. Heterogeneous systems do not impose limits on computational requirements for participants to interact with resources. However, enhancing the efficiency of underlying components is fundamental to encompass both legacy and modern architectures while supporting a wider range of devices;

- **Scalable Candidate Filtering**: for optimized search performance at populated environments. Enterprise architectures have complex network topologies, with segments that can host thousands of devices. Flooding these networks might cause service disruptions when dealing with high traffic. Additionally, integrating efficient service discovery into existing infrastructure may be taken into consideration.

- **Semantic Capability Matching**: to accomplish service specificity and physical requirements. Coupling agnostic service components with a non-agnostic context is a complex problem, especially in CPPS environments.

A summary on how the main works reviewed in this chapter stand against these four properties is expressed in Table 3.1. The disposition includes the criteria listed above, and the assessment follows from what each work reports.

Work	**Discovery basis**	**Decentr.**	**Lightw.**	**Scalable filtering**	**Semantic matching**
Candido et al. [7]	DPWS / WS-Discovery	●	–	–	–
Dai et al. [12, 13]	FBSRT over DPWS	●	–	–	○
Profanter et al. [43]	OPC UA LDS-ME	○	–	○	○
Ismail and Kastner [27]	mDNS + Gateway Bus	●	●	○	–
Cirani et al. [10]	mDNS + P2P, edge LRD	○	●	–	○
Nguyen et al. [37]	WoT TD in JSON-LD	○	○	–	●
Jirkovsky et al. [29]	Shared ontology	–	–	–	●
Lober et al. [34]	Ontology Mgmt. System	–	○	○	●
Dai et al. [15]	Knowledge microservices	○	○	–	●
Ruta et al. [44]	Semantic CoAP / Semantic WoT	●	●	–	●
Fernandez et al. [17]	Compressed identifiers	●	●	○	○
This work	mDNS + CoAP, two phases	●	●	●	●

Table 3.1: Comparison of the main service discovery approaches reviewed in this chapter. Full support (●), partial support (○), not addressed (–).

The comparison clearly illustrates the nature of the gap. Approaches that achieve decentralisation and lightweight operation consistently drop the semantic matching capability, while those that support semantic matching sacrifice either decentralisation or lightness operation. Another noticeable gap is in scalable candidate filtering. Only approaches designed for constrained networks address this challenge, and none of them use semantic information for the purpose.

This trade-off is rooted in the underlying design of existing systems in the sense that semantic matching depends on descriptions that are detailed enough to capture a device's capabilities, but those descriptions are too heavy to include directly in discovery announcements. As a result, consumers must either retrieve descriptions from every responding device, causing discovery costs to increase with the size of the network, or rely on a component that already stores those descriptions, effectively reintroducing a registry.

Analytically, dividing the process into two stages can change this trade-off. A compact summary of a capability may be included in the announcement, making it lightweight enough for every provider to advertise while still being expressive enough to filter out most unsuitable candidates. The full semantic description is then retrieved only for the small number of providers that pass this initial screening. This coarse-grained filtering stage is missing from the decentralised approaches reviewed earlier, and its absence is what makes fine-grained semantic evaluation expensive.

Fernandez et al. [17] come closest to this arrangement by compressing semantic metadata into identifiers that fit an announcement, and the mechanism proposed in Chapter 4 builds on this idea,

extending it from an identifier that can be compared for equality to a summary that can be evaluated against a set of requirements, and adding the second phase that resolves what the summary cannot. Cirani et al. [10] also highlighted that different lightweight protocols, such as mDNS and CoAP, can be used o resolve different parts of the same problem in the scope of service discovery and binding.

Moreover, the IEC 61499 can be considered as a widely recognized suitable tool to enable edge devices modularity. Combined with the developed frameworks and runtime environments in the literature, it is a strong candidate for the development of adaptive mechanism. The ease of developing custom FBs to implement different communication interfaces is a clear advantage for both testing and validating further architectures.

These gaps indicate the need for architectural approaches that combine modular and service-based design principles that are compliant with a standardized discovery protocols, capable of efficiently announce and discover semantic services and devices without compromising any particular specification of the SOA and distributed systems architecture paradigms.

Chapter 4

Proposed Solution

The state of the art developed in Chapter 3 revealed a notable contribution from the IT field that could potentially solve some of the problems that emerged inside the CPPS ecosystem. Solving semantic service discovery in a lightweight implementation will be the primary focus of this dissertation, addressing the flexibility, modularity and agility of SOA enabled industrial environments.

Strongly inspired by the findings of Fernandez et al. [17], Cirani et al. [10] and also the IETF draft that describes the collaboration between mDNS and CoAP protocols [49], the proposed architecture presented in this chapter explores a two-phase distributed mechanism for semantic service discovery in industrial context, which aims to streamline the design and deployment of CPPS, enable real plug-and-produce capabilities and bring the SOA paradigm closer to production environments. The work of Ismail and Kastner [27] further emphasises the importance of accounting for the complex network topologies typically found in enterprise environments. Their findings show that network structure has a direct impact on service discovery traffic and routing, making it an important consideration in the design of scalable and efficient discovery mechanisms.

In a more technical perspective, the designed architecture combines mDNS and CoAP to provide a discovery funnel. First, an inexpensive syntactic pre-filtering on the metadata is performed by mDNS probes, discarding unsuitable resources before a richer evaluation. Then, the CoAP protocol is used to retrieve additional semantic information from a ranked stack of potential candidates. Finally, the service is bound and stored in cache.

The decentralised approaches reviewed in Chapter 3 typically retrieve the service description from every candidate before any filtering or matching procedure. On the other hand, centralised approaches avoid this overhead, but only because all service descriptions are maintained in a single repository. The approach presented with the proposed architecture on this chapter introduces an intermediate step: a compact semantic summary is carried within the service announcement and is used to eliminate most unsuitable candidates before any service descriptions are retrieved. As a result, the complete description is fetched only for the small number of candidates that pass this initial selection.

The two filtering stages also occur at different points in the discovery process. The first is delegated to the network infrastructure, meaning the service category is encoded in the mDNS service type, so providers belonging to other categories simply do not respond to the discovery query. In comparison, other industrial compliant approaches like the legacy WS-Discovery for example, follow a different model in which every device on the local segment responds to the multicast query, and the service category is examined only afterwards by the consumer. Consequently, the consumer must first receive and process every response before it can discard irrelevant services. With the proposed approach, the consumer sees only the subset of providers that advertise the requested category rather than every provider on the network segment.

This stage alone is not sufficient to evaluate detailed service description and a second filtering stage is designed based on evaluating the candidates' potential and retrieving the their semantic description in a ranked order. Fernandez et al. [17] compress semantic metadata into identifiers that are small enough to be included in service announcements. However, these identifiers support only equality matching that would be incorrectly dropped at the first stage. In contrast, the summary proposed here is evaluated against a set of requirements. This enables the first stage to reject a candidate because it lacks the required capabilities, rather than simply because it is associated with a different identifier.

Another important strength of the proposed implementation is that by using widely supported application protocols, the discovery mechanism is naturally designed to function on current industrial network infrastructure. Both mDNS and CoAP protocols work over standard UDP/IP stack and fit well with the managed network environments usually found in industrial settings.

4.1 Rationale and Requirements

The architecture in this chapter is motivated by two complementary observations drawn based on the literature review.

The first observation is quantitative. A WS-Discovery probe and response exchange, based on SOAP-over-UDP, produces messages with approximately 1088 bytes per transaction [27]. A comparable mDNS service announcement uses approximately 300 bytes [27], and a CoAP payload can be confined under 128 bytes by design [48]. That means that, even merging two different technologies to retrieve the full resource information, it will not reach the total payload of a single WS-Discovery transaction. In a populated industrial environment with hundreds of devices, the difference between flooding the network with SOAP probes and using compact UDP-based messages could directly affect the viability of the discovery mechanism in environments where network bandwidth is shared with real-time control traffic.

The second observation is related to the structure of the semantic matching problem itself. In CPPS, service matching happens at two different levels of complexity.

The first level is relatively simple and concerns the general type of capability being requested. For example, the consumer may only need to know whether a device provides a welding, transport, or sensing service. This kind of filtering can be done using a compact identifier, without requiring

full metadata from every available candidate. The second level is more detailed and concerns service compatibility. Even if two devices provide the same type of service, they may not satisfy the same requirements. For example, a welding service may need to support a specific alloy, meet certain accuracy constraints, or be available within a given time window. These requirements depend on structured metadata that cannot be easily represented in short DNS records.

Trying to handle both levels in a single discovery phase introduces clear limitations. Either too much information is placed in mDNS records, making them impractical, or all filtering is delayed until metadata retrieval, losing the chance to eliminate obvious mismatches at an early stage.

These two observations motivate a funnel architecture in which an inexpensive phase eliminates the candidates that definitely do not match the required service, and the expensive phase performs the full semantic evaluation only on the survivors.

From these observations, and also considering the gaps identified in Section 3.3, the following design requirements are derived:

- **R1 - Decentralised Operation**: The discovery mechanism should not depend on any pre-configured central directory. Every device must be capable of announcing its capabilities and responding to discovery probes without prior registration in a dedicated resource manager.
- **R2 - Lightweight Overhead**: The discovery mechanism should minimize the network overhead generated during each discovery cycle, reducing unnecessary traffic as much as possible, especially in densely populated network segments. The pre-filtering phase should also use compact and well defined records.
- **R3 - Semantic Expressiveness**: The architecture should support matchmaking based on functional capability, operational constraints, and also physical parameters. Simple syntactic or type-based matching is insufficient for heterogeneous CPPS environments.
- **R4 - CPPS Alignment**: The service model should be consistent with automation systems, deployment tools and industrial standards, making it deployable using consolidated tools and IDEs.

4.2 Architectural Overview

The proposed architecture organizes the service discovery process into two sequential phases and precedes an optional cache consultation, which must be implemented as an internal component of the service requester agent. Figure 4.1 illustrates the overall structure.

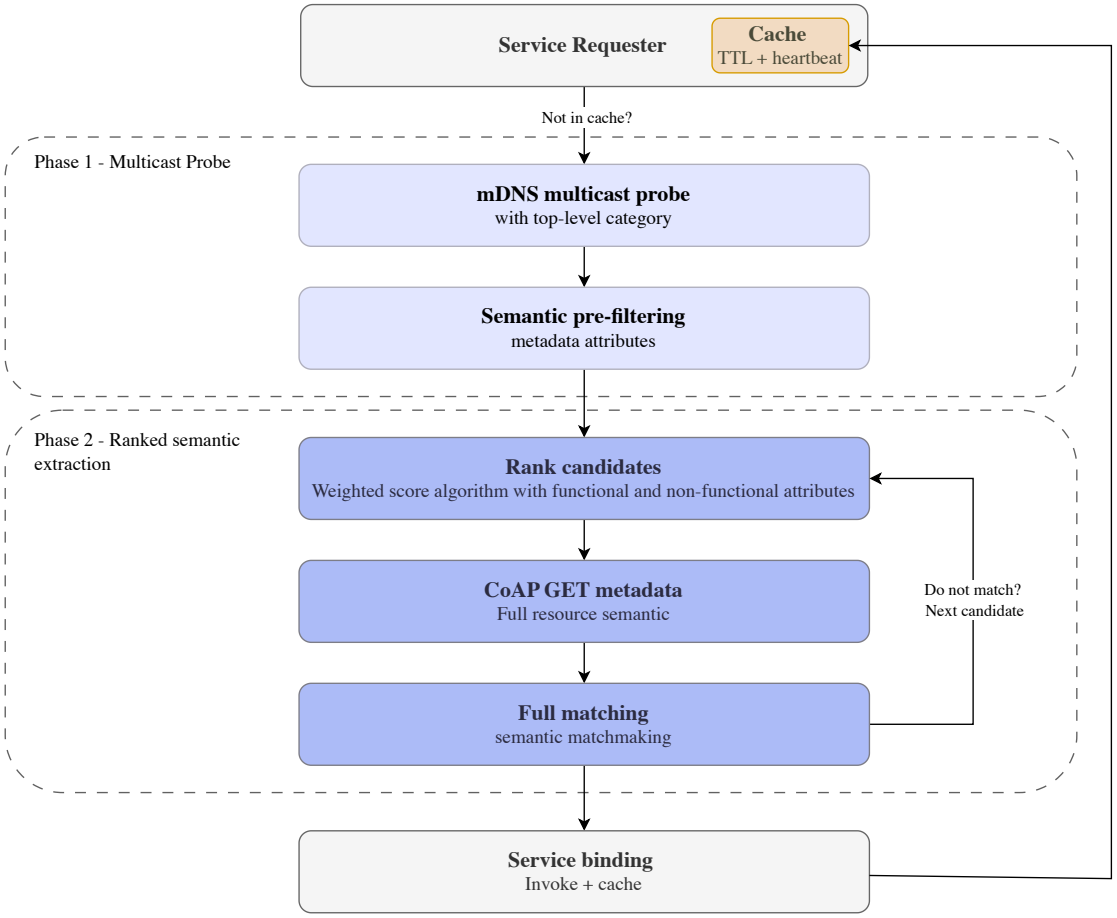


Figure 4.1: Overview of the two-phase semantic discovery architecture. Represents all the components required for the design implementation.

When a service consumer requires a capability, the following sequence occurs:

1. The consumer consults its local cache. If a valid entry exists for the requested capability type, the cached binding is used directly and the sequential discovery phases are not executed. Note that for each service entry, there must be also a reference for its semantic description file, consequently imposing some limitation for memory constrained devices.

2. If no valid endpoint exists in memory, the consumer initiates **Phase 1**: a mDNS probe is sent to the link-local multicast address. Each provider that recognises its own top-level category in the query responds with a mDNS answer containing a structured service name and a TXT record that carries compatibility and operational metadata. The consumer evaluates each response against the request and discards responses that do not pass the pre-filtering phase.

3. The surviving candidates proceed to **Phase 2**. The consumer ranks the candidates using the information already available from the TXT records, then issues CoAP GET requests to each candidate's metadata endpoint in ranked order. Each provider returns a structured semantic metadata document. The consumer performs a full semantic evaluation against the request and selects the first candidate that satisfies all requirements.

4. The selected provider is bound as a service. The consumer optionally stores the binding in its cache with an associated Time-to-Live (TTL) and registers the service endpoint to a heartbeat subprocess.

This sequence respects requirement R2 by concentrating the expensive CoAP exchanges on a reduced candidate set, and requirement R3 by deferring full semantic evaluation to the phase that has access to complete structured metadata.

Note that the service binding and invocation is not specified by this architecture. Further extensions on how to implement service interfaces that could potentially cover different service description languages represents a great research opportunity and goes beyond the scope on this dissertation. Nevertheless, this architecture provides a strong support for service description within the CoAP semantic request.

4.3 Phase 1: Semantic Pre-filtering

The first phase addresses two questions simultaneously: which devices on the network segment belong to the requested service category, and which of those are suitable for the second exchange. The first is determined by the service type under which a provider advertises itself, while the second relies on the metadata included in the announcement. The following two subsections describe these mechanisms in turn, and the final subsection explains how the consumer combines them to identify the most suitable candidates.

4.3.1 mDNS Announcement Model

Each service provider announces its capabilities using the standard mDNS/DNS-SD announcement model [8, 9]. A service instance is described by four DNS record types:

- A **PTR record** that maps a service type to a service instance name, enabling consumers to discover all instances of a given type.
- A **SRV record** that provides the hostname and port of the provider.
- An **A/AAAA record**, IPv4 and IPv6 respectively, that resolves the hostname to an IP address.
- A **TXT record** that carries arbitrary key-value metadata about the service instance.

The service type field, which appears in the PTR record, is defined in the DNS-SD specification as a string of the form `<Service>.<Domain>`. A first design decision concerns how much of the service metadata to carry in this field. It is tempting to encode the full capability hierarchy inside the `Service` field, as in `_actuator._pneumatic._valve._coap._udp`, and let consumers narrow their probes by querying deeper levels. This does not work with the standard mDNS when considering semantic matching capabilities. The DNS resolution model

matches names by exact string equality, with no prefix, wildcard, or partial matching. A probe for `_actuator` would never reach a provider that registered `_actuator._pneumatic._valve`, since the two strings differ.

Any apparent hierarchy in the service name is not something the protocol traverses and is only possible through non-standard extensions to the protocol. An option this work deliberately avoids to preserve compatibility with legacy mDNS stacks. The same objection applies to encoding partial queries in a TXT record and carrying them in the additional section of an mDNS query, as proposed by Liao et al. [32]: the matching still relies on behaviour that standard resolvers does not implement and is also seen as malformed stack by some sniffing tools, such as Wireshark.

For this reason, the architecture places a top-level category in the service type and defers finer discrimination to the TXT record. All devices belonging to a category register an identical `Service` string, which is used for constructing a mDNS query (Figure 4.2). Each individual service instance is then distinguished by a unique instance identifier (a UUID or a stable hostname) carried in the DNS-SD instance name, as the specification already prescribes (Figure 4.3).

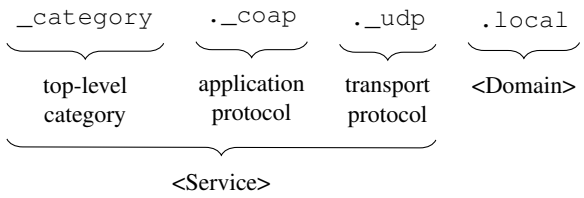


Figure 4.2: Structure of the proposed mDNS service query (PTR record).

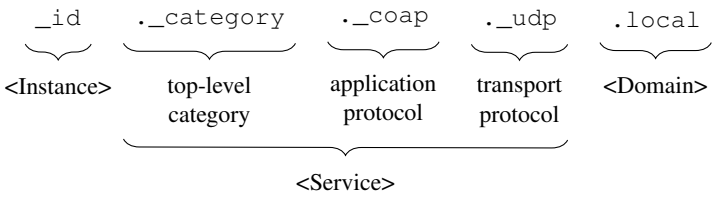


Figure 4.3: Structure of a PTR response derived from a previous query.

Making the application protocol explicit is also a design choice that not only allows the extension of the proposed model but also helps to distinguish service providers that are not supported by the architecture.

Choosing which dimension to promote to the top-level category is not an arbitrary decision, and the architecture does not fix it a priori. Functionally, its structure acts as an index over the device population. A good category has low cardinality, divides the population well and is known to the consumer at query time. Category dimensions can include functional type (sensor, actuator, controller, etc.), location, or a combination of both, for example. Because the name is matched exactly, promoting more dimensions improves selectivity per probe but, in the absence of wildcards, forces the consumer to issue one probe per combination.

A single top-level category is enough for the environments this dissertation targets, but the design does not depend on that choice. In a plant with many devices, or with many different kinds of them, it can make sense to add more levels to the name. So the depth of the name is not fixed by the architecture - it depends on engineering design.

4.3.2 TXT Record Metadata Structure

The TXT record carries the metadata required for pre-filtering and for Phase 2 candidate ranking. Given the practical size constraint of approximately 200 bytes recommended by RFC 6763 [8], the TXT record is designed to encode only the attributes necessary for the pre-filter and ranking decisions, leaving detailed capability parameters to the CoAP metadata document.

The proposed TXT record structure includes the following fields:

- `version`: The version of the ontology vocabulary used to encode semantic description into CoAP metadata documents. This field allows the consumer to determine whether it can interpret the metadata document returned by the provider.
- `model`: The brand and model identifier of the device. Some consumers can only interact with specific vendors or hardware revisions, for driver or protocol-conformance reasons. This field lets such a consumer rule out incompatible providers during the first phase.
- `path`: The URI path of the CoAP endpoint serving the semantic metadata document (e.g., `path=/well-known/meta`).
- `zone`: A logical location identifier corresponding to the physical zone or production line where the device is deployed (e.g., `zone=line3`).
- `load`: Indicates, qualitatively, the current load on the device. It can be a categorical value such as `low`, `moderate`, or `high`. If a resource is shared among multiple consumers, this field can be used to avoid overloading a device that is already heavily utilized, even though it may still be compatible with the request.
- `setup`: Represents the setup time, context switching time or any other temporal constraint that may affect the service's agility. Between two otherwise equal candidates, the one that can start sooner is preferable.
- `health`: A numeric value representing the current health of the service. A normalized indicator of recent reliability can vary between two fixed and well defines values, such as 0 and 1.

What the proposed architecture actually requires is only that the TXT record carries enough information for the consumer to reach the metadata endpoint and to order the surviving candidates. It means that a deployment with different priorities could drop some of these fields, add others, or

rename them without breaking the discovery mechanism. An example TXT record for a welding robot, following the fields described above, is shown in Figure 4.4.



Figure 4.4: Dissection of an mDNS response: Example of an additional section with expanded TXT record.

4.3.3 Pre-filter Evaluation

After the mDNS syntactic matching, the consumer must eliminate any candidate entity that does not meet the mandatory conditions outlined in the request. This strictness is not compensatory and must be exclusive, which means the fields examined at this stage are not quality indicators but rather fundamental prerequisites for continuing the information exchange or for the final matching process. In other words, a failure to meet these requirements does not render the service provider inferior, but rather unsuitable for the intended purpose, making any further communication with it unnecessary.

The threshold against which each condition is tested belongs to the requester and, for that reason, the criteria must be defined and deployed to the consumer at the design phase and there must be a consensus, or at least a shared expectation, between what the provider advertises and what the consumer expects. The same field can be a strict requirement for one consumer and irrelevant to another: for example, a consumer physically restricted to a single production line area treats a `zone` mismatch as fatal, while a mobile consumer able to operate anywhere may

ignore this field entirely. Another example is a consumer that requires a secure CoAP connection, which is a condition that may be critical for some applications but irrelevant for others.

The development of Algorithm 1 states the evaluation and it is non-compensatory by construction. It means that a single failed condition drops the candidate and this is what separates the pre-filter from the ranking that follows, where a weak value in one criterion can be offset by a strong value in another.

Algorithm 1 Phase 1 pre-filter, consumer side

Require: request q, set of mDNS responses R
Ensure: candidate list C

$C \leftarrow \emptyset$
for all $r \in R$ **do**
  $viable \leftarrow$ **true**
  for all hard condition $c \in \text{hard}(q)$ **do**
    if $\neg\,\text{advertises}(r,c) \lor \neg\,\text{satisfies}(r,c)$ **then**
      $viable \leftarrow$ **false**
      break ▷ missing or mismatched field is enough to discard
    end if
  end for
  if *viable* **then**
    $C \leftarrow C \cup \{r\}$ ▷ append candidate to survivors
  end if
end for
return C ▷ hand the survivors to Phase 2

Finally, the output of Phase 1 is a list of candidate providers that passed both mDNS syntactic matching and the pre-filter evaluation, containing the TXT metadata already retrieved at announcement time, which the ranking stage reuses at no additional network cost.

4.4 Phase 2: CoAP Semantic Retrieval and Matchmaking

Candidates that pass the first phase are not guaranteed to satisfy the service request, they are simply considered promising enough to justify further evaluation. The purpose of the second phase is to make that final evaluation and the final selection. Since each candidate requires an additional network exchange to retrieve its complete semantic description, the order in which candidates are evaluated directly affects the overall communication cost. For this reason, the ranking strategy is presented before the retrieval process.

4.4.1 Candidate Ranking

Before issuing any CoAP request, the consumer ranks the surviving candidates from Phase 1 using the information already available in their TXT records. The purpose of ranking is to increase

the probability that the first CoAP request returns a satisfactory match, minimizing the total number of metadata retrievals required and, consequently, reducing the total network overhead of the semantic discovery process.

Since a provided service may not fully satisfy all the requirements of a requester, the ranking algorithm must be able to express a preference for a candidate that mostly fits all the quality criteria in a compensatory way. A requester that is too strict about a parameter, such as timing or health, may heavily penalize a candidate with a weak performance in those dimensions.

The ranking score $s(r)$ assigned to candidate r, Equation 4.1, can be computed as a Simple Additive Weighting model, which compensatorily evaluates the fields carried in its TXT record:

$$s(r) = \sum_{i \in \{z,l,s,h\}} w_i \, \phi_i(r), \qquad \phi_i(r) \in [0,1], \qquad \sum_i w_i = 1 \tag{4.1}$$

where w_i indicates the weight for criterion i, and $\phi_i(r)$ a normalized score for candidate r in that criterion.

Since different criteria may be expressed in incomparable forms, units and scales, the ranking algorithm must normalize them to a fixed scale before combining them. It means that each normalised criterion $\phi_i(r)$ is defined so that 1 is the most desirable value and 0 the least. The same is applied for all the weights w_i and consequently, the final score $s(r)$ is also normalized to the same range.

As an example, consider the setup time. Among the surviving candidates, let $t_{\min}$ and $t_{\max}$ be the shortest and longest advertised setup times reported through all the survivors TXT records. A candidate r with setup time $t(r)$ contributes as stated in Equation 4.2

$$\phi_s(r) = \frac{t_{\max} - t(r)}{t_{\max} - t_{\min}} \tag{4.2}$$

A binary criterion, such as zone co-location, takes only the values 0 or 1 directly. A missing field, as discussed earlier, is assigned the worst-case value 0, so depending on the weight w_i for that criterion, an undescribed candidate sinks to the bottom without being discarded.

It is important to note that this model imposes an assumption that all the criteria are independent, which means that, for fields that may be strongly correlated, such as `load` and `setup`, the ranking algorithm will penalize a candidate twice for the same underlying condition. In other words, a device that is heavily loaded also tends to report a longer setup time, and both conditions should be merged to a single criterion before penalizing a candidate. Methods that capture these interactions exist, for example the Choquet integral [22], used to assign weights to every possible combination of criteria, ideal for non-independent sets. However, choosing the lighter model here is a deliberate trade-off and although being less expressive, it should be sufficient to evaluate the proposed model. Moreover, implementing the right method is an open design decision, which may justify the use of more complex algorithms.

4.4.2 CoAP Semantic Metadata Document

After ranking the potential candidates in a priority order, the service requester should proceed with the real semantic evaluation. This step requires a richer set of metadata and a more structured representation of the service. In that sense, to promote a more standardized and lightweight approach, the metadata document should be encoded in JSON-LD. The choice of JSON-LD is motivated by its native compatibility with CoAP's content format negotiation, its human readability, and its ability to carry semantic annotations. Moreover, being a JSON-based format, it can be read by the generic JSON parser a device is likely to already have, without requiring a dedicated RDF parser as heavier serializations such as RDF/XML or Turtle would.

The structure of this document does not need to be invented from scratch. The World Wide Web Consortium (W3C) WoT TD [30] already defines a JSON-LD information model for describing devices and their interactions. A basic TD model organizes the device's metadata into a key:value structure with four mandatory fields at the top-level, which are described below:

- **Context**, defines the namespaces used in the document, allowing the consumer to extend and interpret additional tags. The namespace is linked to a vocabulary prefix which identifies whether the tag is a custom extension or a standard term.
- **Title**, a human-readable name for the device. In the design proposed here, this is set to the device's hostname, the same name returned inside the mDNS SRV record but without the domain suffix.
- **securityDefinitions**, a catalog of security schemes that are required to interact with the device's resources. Each scheme is a set of parameters defined by the standard WoT vocabulary, and is mapped to a tag which is then referenced by the `security` field at different levels of the document.
- **Security**, which selects the security scheme to be used at the current scope. The selection is made by reference to a tag defined in the `securityDefinitions` section. The security scheme can be defined at the top level, applying to all interactions, or at a more granular level, applying only to a specific form.

Before describing the interaction affordances, it is worth clarifying what a Thing represents in this architecture, since the term is often used loosely. In the WoT model, a Thing is the abstraction of a physical or virtual entity, and not necessarily a single service. The decision adopted here is that a Thing corresponds to a device, physical or virtualized, and not to an individual service. A device that offers several capabilities is described by a single TD, in which each capability is modelled as an Action. This has a direct consequence for the discovery mechanism: a device serves always the same TD, regardless of which service motivated the search. The consumer locates the device by its category in Phase 1, and the concrete service is selected at the Action level, inside the TD retrieved in Phase 2. In the example that follows, the device `weldbot-07` is typed as a `WeldingRobot` at the Thing level, while the welding capability it exposes is typed as a `WeldingService` at the

Action level. This design choice also enables inadvertent discovery of additional services with no further packets exchange, since the TD document should describe all available services running on a target device (Thing).

A TD document can also carry interaction affordance information, which are the actual resources that the device exposes to the consumer. Figure 4.5 shows the entity relationship between each affordance object and the Thing entity, and how these objects retain data type information, semantic meaning and entry point.

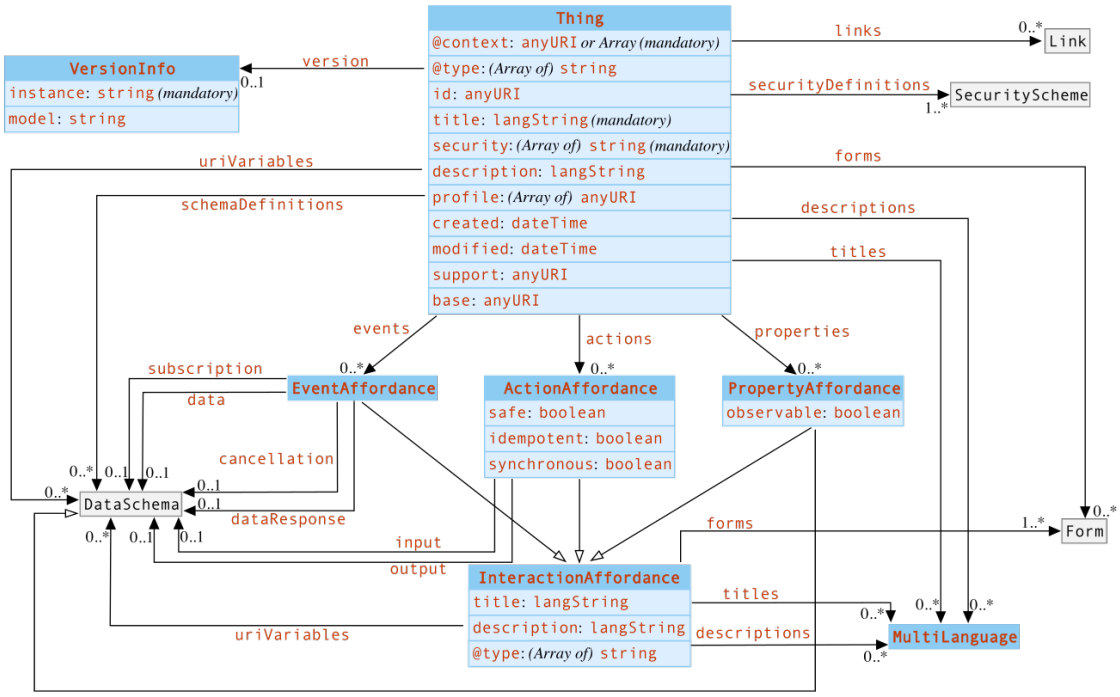


Figure 4.5: Class diagram of TD core vocabulary. From [30].

The `PropertyAffordance` describes the service state, allowing the consumer to read informational data by exposing an access endpoint. The `ActionAffordance` describes methods that can be remotely invoked and should explicitly declare input and output parameters and data types, as well as their synchrony mode. Finally, the `EventAffordance` describes asynchronous notifications that the device can push to the consumer [30].

For the discovery mechanism architecture proposed in this dissertation, the TD document must carry a static description of the service and an entry point to which the requester should bind to send a request. For this reason, two deliberate simplifications are made to keep the mechanism lightweight: The first is that descriptive semantic attributes of a service should be embedded directly into the TD document, with no endpoints to fetch the properties, as the standard recommends to do. This would introduce an unnecessary network interaction, which the architecture must avoid at this stage. Second, a service should not directly expose its internal methods, instead, it must provide a connection interface through which the requester can permanently bind to it. Although

the binding process might involve invoking a specific method exposed to the consumer, this abstraction guarantees that the consumer interacts with services via a standardized interface, which may vary depending on the underlying implementation.

Finally, domain-specific attributes should be added to the core of the TD document through the extension mechanism provided by the `@context` field. In the context of this architecture, declaring a custom namespace allows the definition of types that can be shared between the consumers and reused by the providers, just like a standard vocabulary. This is strictly necessary when parsing JSON structures during the service evaluation, and a consumer that does not recognize these terms can still process the rest of the document, since a TD processor ignores annotations it does not understand. Listing 4.1 shows an example considering all the simplifications stated above while constructing a metadata document for a welding robot (`weldbot-07`).

Listing 4.1: Simplified Thing Description document. The `input`, `output` and `forms` fields are shown abbreviated.

```
{
  "@context": [
    "https://www.w3.org/2022/wot/td/v1.1",
    { "cpps": "https://example.org/cpps#" }
  ],
  "id": "urn:uuid:weldbot-07",
  "title": "weldbot-07",
  "@type": "cpps:WeldingRobot",

  "securityDefinitions": {
    "nosec_sc": {
      "scheme": "nosec"
    }
  },
  "security": "nosec_sc",

  "cpps:certifiedMaterials": [
    "AA2024",
    "S235"
  ],

  "cpps:positionalAccuracy": {
    "value": 0.25,
    "unit": "mm"
  },

  "actions": {
    "weld": {
      "@type": "cpps:WeldingService",
      "cpps:fbType": "WeldControl_SIFB",
```

```
      "synchronous": true,
      "input": {...},
      "output": {...},
      "forms": [...]
    }
  }
}
```

This document encodes attributes that are structurally impossible to represent in a TXT record: a list of certified material alloys and typed physical quantities with explicit units, such as the positional accuracy. This is exactly what justifies the two phases. In Phase 1, `weldbot-07` passed the filter simply by being an actuator in the right zone - all the TXT record could tell the consumer. Whether it can actually weld the requested alloy, within the required positional accuracy, is only visible here, in the metadata document. A second welding robot in the same zone could pass the identical Phase 1 filter and yet be certified for a different set of alloys, a distinction that no compact DNS record could have carried.

4.4.3 Semantic Matchmaking

Upon receiving the metadata document, the consumer performs a full semantic evaluation against the service request. The evaluation checks functional compatibility (capability type, certified parameters, operational envelope) and the physical constraints that the TXT record could not express, such as positional accuracy. If the candidate satisfies all requirements, it is selected and the discovery cycle terminates without querying the remaining ranked candidates.

If the candidate does not satisfy the requirements, the consumer proceeds to the next ranked candidate. This ordered evaluation means that in the best case - where the top-ranked candidate matches - only a single CoAP request is issued after Phase 1. In the worst case, all candidates are evaluated. The expected number of CoAP requests is therefore bounded by the pre-filter quality and the ranking accuracy.

If the metadata document is annotated with a vocabulary that supports it, the matchmaking step can go beyond exact comparison and reason over the `@context`. Were the vocabulary to declare that `AA2024` is a kind of aluminium alloy, a consumer asking for any aluminium alloy service would match a provider certified for `AA2024` without an exact string match. This capability is a consequence of the chosen vocabulary rather than of the discovery mechanism, and how much inference to support is a trade-off between matchmaking accuracy and the computational cost it places on constrained devices.

4.5 Service Binding and Caching

The discovery process ends when a provider has been selected, but the consumer still has to reach it, and repeating the whole funnel for every subsequent call would defeat the purpose of having

measured its cost. This section describes how the selected provider is bound and how the result is retained.

4.5.1 Service Binding

Once a matching provider has been selected, the consumer binds to it using the entry point and the interface descriptor carried in the TD retrieved during Phase 2. At this level, the architecture defines only the contract of the binding: the consumer holds an endpoint and a description of the expected inputs, outputs and invocation mode, and establishes a connection to that endpoint. How this connection is concretely realised is deliberately left unspecified, and should not be coupled to any particular industrial protocol or execution standard. Different deployments may bind through a plain CoAP client, a wrapper over an existing middleware, or a native runtime construct, without any change to the discovery mechanism that precedes it.

As a concrete example, and to ground the mechanism in existing work, the binding can be realised through the SIFB construct of IEC 61499, an approach already demonstrated in the literature [14]. In this case, the endpoint obtained from the TD parameterises a SIFB that mediates the communication between the consumer's application and the remote service, and the discovery result becomes directly actionable inside an IEC 61499 runtime. This is presented as one possible instantiation rather than a requirement: the SIFB illustrates how a discovered service is turned into a running binding, but the architecture does not depend on it, and an implementation targeting a different execution environment would substitute its own binding construct while reusing the same discovery output.

4.5.2 Cache Model

To avoid repeating the full two-phase discovery process for recurring requests, the consumer maintains a local discovery cache. This cache is intentionally simple: it stores resolved bindings in memory for as long as the service browser is running. There is no need to persist it, as the network itself is the source of truth for service availability. If the cache is lost when the process stops, the consumer can simply perform discovery again. In fact, persisting the cache across restarts could be counterproductive, as it might restore bindings that point to devices that have since left the network. For most consumers, a simple in-memory dictionary keyed by the normalised request is sufficient. Applications that perform a large number of discoveries may instead use an eviction strategy such as Least Recently Used (LRU), but the cache remains an in-memory structure rather than being stored in a database.

Each cache entry maps a normalised request, consisting of the requested category and its mandatory conditions, to the endpoint of the provider that satisfied it. Using the normalised request as the cache key ensures that a cache hit occurs only when a new request has the same required constraints as the original one. Requests with different mandatory conditions are treated as new discoveries, preventing incorrect results from being returned from the cache.

Each cache entry carries an associated TTL. Choosing this TTL interacts with the underlying mDNS layer, and the distinction has to be made explicit. The mDNS records themselves carry their own TTLs, defined by the protocol, which govern how long a discovery response is considered valid at the name-resolution level [9]. The application-level cache proposed here sits above that: on a cache hit the consumer skips both phases entirely and never emits a probe, so the mDNS TTL does not expire the cached binding. It does, however, provide a sensible upper bound for the application TTL, since it makes little sense to trust a resolved binding for much longer than the discovery protocol itself would consider its records fresh. The TTL is otherwise set conservatively based on the expected stability of the resource: physical actuators in a fixed production line are assigned longer TTLs than software services or mobile assets.

A passive expiration mechanism removes entries whose TTL has elapsed. In addition, because the service browser listens to the mDNS traffic continuously while running, it can invalidate an entry proactively when the corresponding provider announces its departure through an mDNS goodbye packet, a record advertised with a TTL of zero when a device leaves the network gracefully [9]. An active heartbeat mechanism has to be proposed using the CoAP Observe option [23], in which the consumer registers an observation on the provider's metadata endpoint after binding. If the provider departs abruptly, the CoAP Observe notification ceases and the consumer receives a Reset (RST) message or a timeout, triggering cache invalidation and a new discovery cycle.

Chapter 5

Implementation

5.1 Overview

The architecture described in Chapter 4 was implemented as a self-contained software package that realises the two-phase discovery funnel and the provider that answers it. The implementation is guided by a small set of requirements that follow from the architecture and from the intended use of the prototype:

- **Fidelity to the two phases.** The mDNS pre-filtering of phase one and the CoAP metadata retrieval of phase two must remain distinct and observable, so that the behaviour of each of them can be reasoned about on its own.
- **Configurability.** A device must be described entirely by external configuration, with no device-specific logic in the code, so that different devices are a matter of different descriptions rather than different programs.
- **Independence from deployment.** The package won't take assumptions about how it will be run or evaluated. It will only exposes the discovery mechanism thought objects and methods.
- **Standard compliance.** The implementation must stay within standard mDNS and CoAP behaviour, so that the mechanism interoperates with unmodified stacks, as argued in Chapter 4.

Not everything in the architecture was implemented, and the omissions are deliberate. The service binding is left out because the architecture does not specify it. In other words, what the funnel produces is an endpoint and an interface descriptor, and turning that into a running connection belongs to whatever execution environment sits underneath. Implementing one particular binding would describe that environment rather than the discovery mechanism, and would tie the prototype to a choice the architecture deliberately avoids.

The cache is left out for a different reason. A cache system alone, that expires only by a TTL is not a faithful cache for this problem, because a device can leave the network at any moment and

a purely time-based entry would keep pointing at it until it aged out. Doing it properly requires a browser that outlives an individual search together with a mechanism to detect departures, like a heartbeat mechanism, which forms a separate piece of work with its own design. The cache is therefore described in the architecture but not carried into this prototype.

5.1.1 Chosen Language

The architecture was implemented as a Python package, so that the mechanism could be exercised on real hardware. The honest reason of choosing Python at this stage is the development speed in the sense that the mechanism may need to pass through several rounds of design changes, and rewriting a class is cheaper than rewriting a build. There are also available mature libraries to develop the architecture, discarding any side development.

Although existing constraints in the Python environment, such as large memory footprint and global interpreter lock, the deployment of this architecture to a constrained device would not likely be affected by them, since the cost of a discovery is dominated by waiting for the network, not by computing. The pre-filter is a handful of string comparisons and the ranking algorithm chosen for this implementation is a simple weighted sum. Whatever the language, the requester spends almost all of its time executing and waiting for IO operations.

5.1.2 Libraries

Three libraries carry the protocol work, one for each concern: the mDNS layer of phase one, the CoAP layer of phase two, and the configuration format.

python-zeroconf implements mDNS and DNS-SD [8, 9]. On one hand, it is a mature, actively maintained implementation of mDNS and DNS-SD for Python. Its adoption in production software and continued maintenance make it an appropriate implementation for the service discovery mechanism evaluated in this work. On the other hand, it presents a mismatch of interaction model. The library is built around continuous browsing, in which a browser is started, remains active, and issues callbacks whenever a service appears, changes, or leaves the network. The mechanism proposed here does the opposite, issuing a single targeted probe for a category, waiting for a bounded interval, and then deciding. The gap was bridged by wrapping the browser in a probe cycle that starts it, waits, collects the responses, and stops it, so that the rest of the code sees a request-response interaction even though the library underneath implements a subscription model.

aiocoap implements CoAP [48]. It is the maintained CoAP library for Python that covers what the second phase requires, so the choice was effectively made by the ecosystem. It is asynchronous only, which settled the concurrency model for the whole package. Once the CoAP side is built on `asyncio`, the mDNS side and the funnel that drives them are written the same way. This was no loss, since a discovery is mostly waiting, and an asynchronous model expresses waiting on several things at once more naturally than a threaded model would.

One characteristic of aiocoap did require care. Its network transports are selected according to the platform, and they do not all support the same options. In particular, the transport that binds a

single socket to a dual-stack any-address is not available on every operating system, and in which it is absent the fallback transport refuses an any-address bind altogether.

PyYAML reads the configuration files. YAML Ain't Markup Language (YAML) was preferred over JSON because these files are written and edited by hand, frequently contain numeric values, and benefit greatly from comments, which JSON does not allow. The cost of this choice is that the parser performs no validation: a misspelled key is not reported as an error but silently ignored, and a device configured with such a key comes up quietly incorrect. YAML also introduces a typing subtlety, discussed in Section 5.3, since it returns a number where the underlying protocol carries only text.

5.2 Anatomy of a Discovery

Along with the practical choices that are strongly related to the language, it is also important to describe how the pieces fit together. That path is the funnel of Chapter 4, but now made of objects. A configuration object, together with a configuration file, defines what exists, a provider announces it, and a search engine is finally capable of finding it. Figure 5.1 maps the parts and the two exchanges that connect them, and the subsections that follow walk the same path in words, from the description of a device to the choice of a candidate.

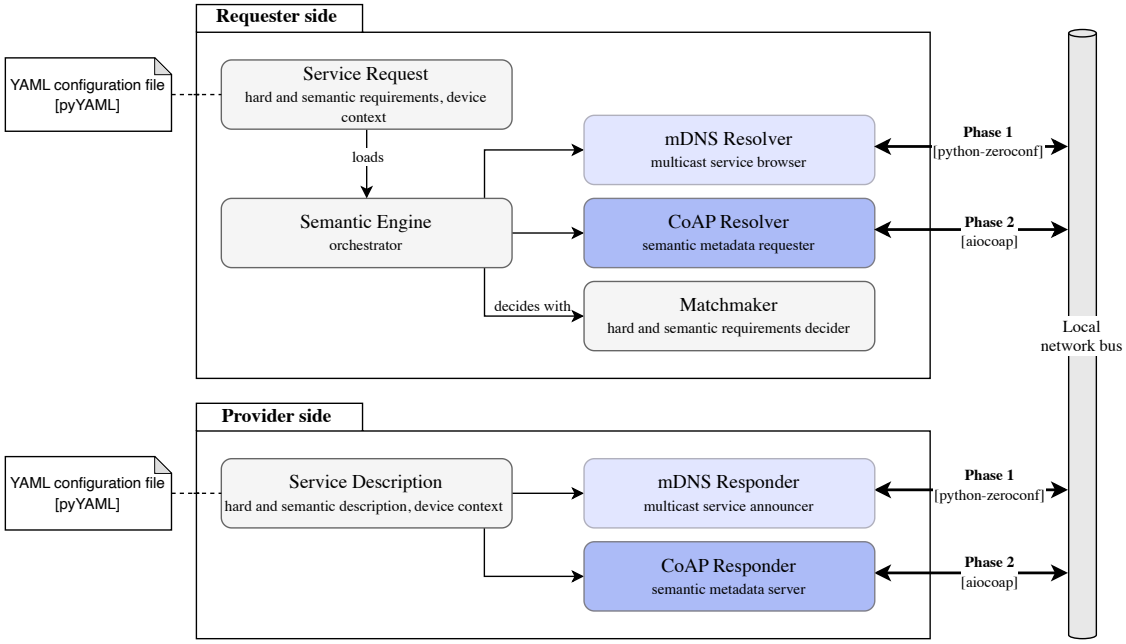


Figure 5.1: Component view of the implementation.

Two things are worth reading off the figure. The first is that the requester side and the provider side share no code and hold no reference to one another. They meet only over the network, once in each phase. The second is that the same libraries are reused on both sides, because both speak the

same protocols and the only difference is the direction: one announcing where the other probes, one serving where the other fetches.

5.2.1 Device Configuration

A device and its capabilities are described in a YAML file, and a process that reads it becomes that device. This keeps device-specific detail out of the logic and confines it to data, which is what lets a single program stand in for an arbitrary population of different devices.

At the provider's side, the configuration file carries what the provider needs to execute its role: the instance identifier, the top-level `category`, the TXT record, the CoAP port, and the TD to be served in the second phase. The document is embedded in the same file rather than kept beside it, so that a device is a single artefact.

At the requester's side, the configuration file describes the target `category`, which travels in the mDNS name, the `hard` conditions that are the viability constraints tested against the TXT record in phase one, the `weights` that drive the ranking and the `semantics`, which are the requirements evaluated against the TD in phase two. There is also a `context` field, which is not a criterion but the requester's own location, used to evaluate its co-location against a provider.

The weights are normalised when the file is loaded, so that they sum to one. This is not merely a convenience: the ranking equation in Chapter 4 assumes it, and without it the score leaves the $[0, 1]$ range and ceases to be comparable between requests. The practical effect is that a file may express priorities as plain proportions, so that writing `2/1/1/1` or `0.4/0.2/0.2/0.2` amounts to the same thing.

5.2.2 The Provider

A provider does not execute any service at this stage. Since the architecture does not specify binding or invocation, a device in this implementation is essentially an announcer of descriptions: it exists on the network and hands over its TD to whoever asks. It has two faces and one coordinator, shown on the right of Figure 5.1, and each face corresponds to one phase.

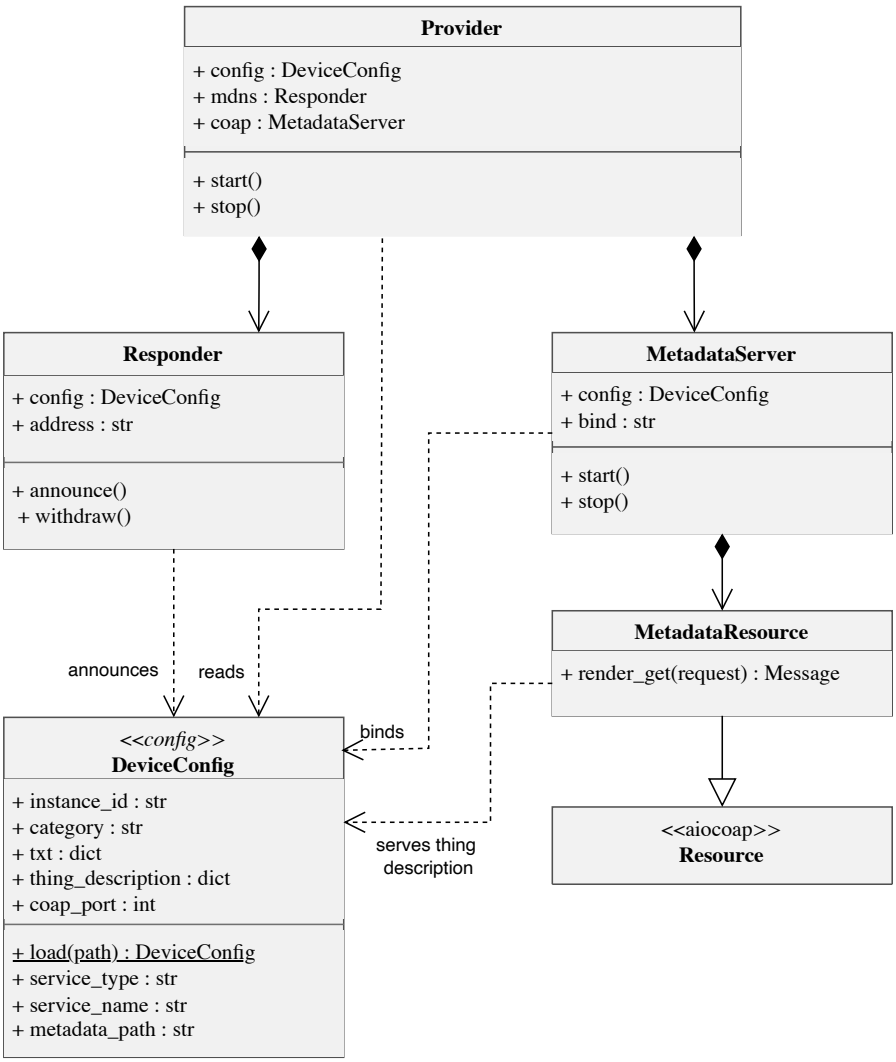


Figure 5.2: Service provider's side class diagram.

The `Responder` is the presence, and it belongs to phase one: it registers the service in mDNS with the category in the name and the configured fields in the TXT record, and it deregisters on the way out, so that the device graciously leaves the network rather than simply vanishing. The `MetadataServer` is the CoAP side, belonging to phase two, serving the TD at the path the TXT record advertises. That path is read from the TXT record itself rather than configured twice, so that what the device announces is, by construction, what it serves.

The document is returned with content format 432, the value registered by the Internet Assigned Numbers Authority (IANA) for `application/td+json`. Using it rather than plain `application/json` has no cost and is done by convenience. The document is serialised once at start-up, since it does not change, so that answering a request is a matter of IO alone.

One detail of the provider worth recording is that the address a device announces and the address on which it listens are two separate settings, and nothing inherently forces them to agree. A provider that announces one address while listening on another is discovered correctly and then

refuses every connection. The two are therefore tied together by the implementation: a device instructed where to listen, but not instructed what to announce, announces the address on which it listens, and a device given two settings that disagree is warned of it.

5.2.3 The Search Engine

The engine is not a consumer. It is the search machinery that a consumer uses, and what becomes of the candidate it returns is the concern of whoever called it. This mirrors the separation the architecture draws between the discovery mechanism and whatever invokes it, and the distinction is reflected in the naming.

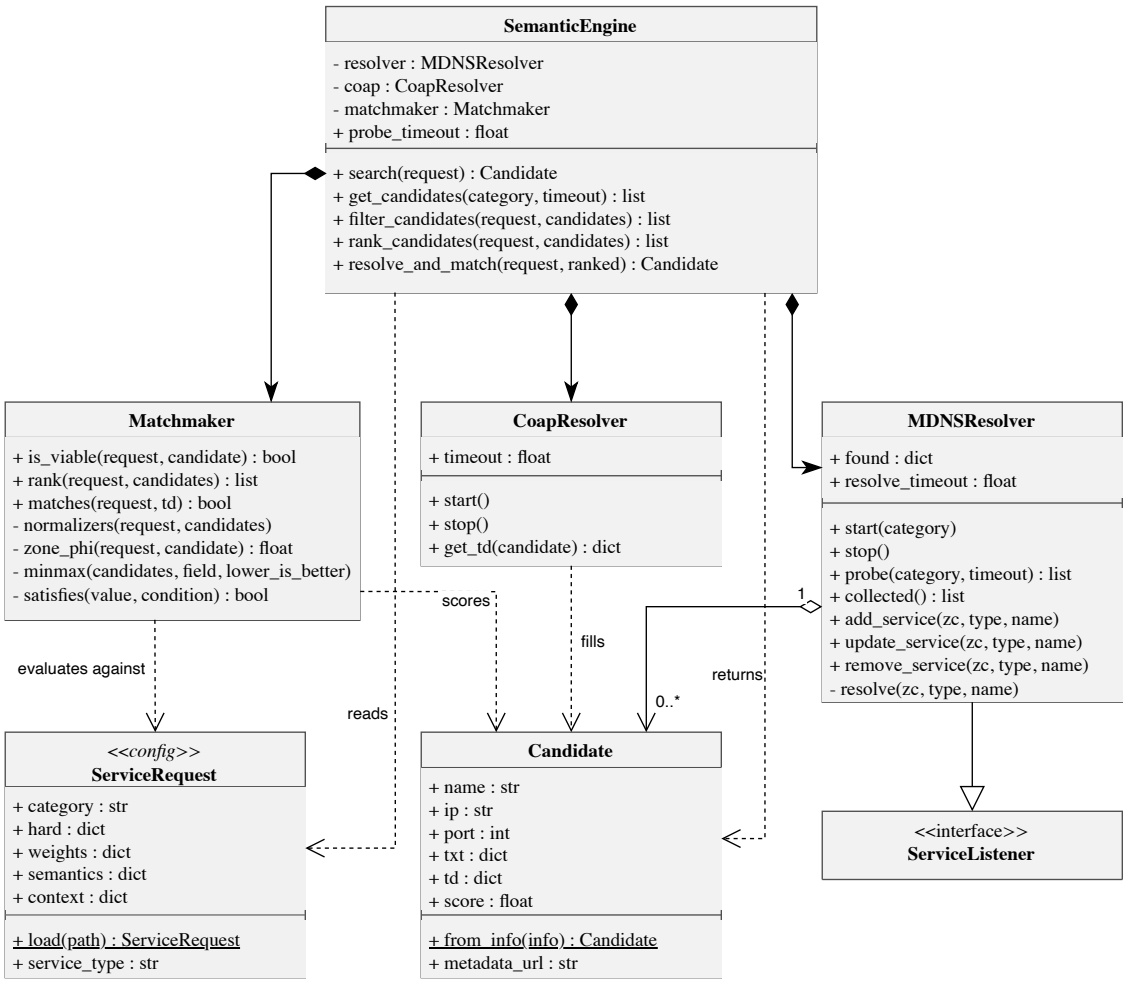


Figure 5.3: Semantic Search Engine class diagram.

`SemanticEngine` orchestrates and delegates. Its `search` method reads from top to bottom as the funnel itself: probe the category, filter the responses, rank what survives, and only then spend network effort fetching documents, in the order the ranking decided, stopping at the first that satisfies the request. The intermediate steps are exposed as well, which allows each phase to be driven and observed independently rather than only through the combined operation.

The `MDNSResolver` owns everything that touches zeroconf. It implements the library's listener interface, through which the three callbacks for services appearing, changing, and leaving arrive, and it keeps what it finds in a dictionary keyed by instance name, which absorbs the repeated announcements mDNS makes and drops entries when a goodbye arrives. It resolves rather than merely records, because a PTR answer alone carries only the instance name; the SRV, TXT, and A records that make a candidate usable require a follow-up request, which is issued as a task from the callback and awaited when the probe closes.

The resolver decides nothing, and this is deliberate for two reasons. The first is separation of concerns, which means a resolver that filtered would have to know what a request is, and would cease to be a resolver. The second is observability: if candidates were discarded as they arrived, the number that responded would be lost, and that number, set against the number that survived, is precisely what quantifies the selectivity of the first phase. The same reasoning applies to `CoapResolver`, which fetches a document and hands it over without forming an opinion about it.

All the deciding is concentrated in the `Matchmaker`, in three methods that correspond to three different decisions. `is_viable` is the non-compensatory conjunction over the TXT record, where a single failed condition is enough to discard and a missing field fails in the same way as a wrong one. `rank` is the compensatory weighted sum over the same record. `matches` is the semantic evaluation against the TD. Keeping them apart, under those names, lets the code express directly that the two phases rest on two opposite decision models, a claim made in Chapter 4 that would be invisible were the three folded into one.

5.3 Decisions Forced by the Implementation

Some aspects of the design only became apparent once there was code to run, and they changed the design itself rather than only its expression.

The first is that DNS-SD TXT records carry text and nothing else. A configuration file may write `version: 1` and intend the number, and a request may do the same, but what arrives from the network is the string `"1"`, and comparing the two directly fails and discards a device that was in fact compatible. Hard conditions over the TXT record are therefore textual comparisons, with both sides converted to text before testing. This is less a workaround than an acknowledgement of what the transport actually is.

The second concerns the ranking. The normalisation described in Chapter 4 is min-max, and min-max is relative to a set: the best setup time among the candidates is only known once every candidate has been seen. A method that scored one candidate in isolation could not compute it. The ranking therefore operates on the list of survivors and returns it ordered, assigning each score along the way, which is a different shape from the one the equation suggests when read in isolation.

The third gave rise to the `context` section of the request. The zone term of the ranking scores co-location between the candidate and the requester, which requires knowing where the requester itself is. That is not a criterion and does not belong among the constraints, so it is held in a section

of its own. Without it the term is undefined, which is easy to overlook on paper and impossible to overlook in code.

The fourth is smaller but of the same nature. The setup field was first expressed as a category, with values such as `fast` and `slow`, which reads naturally but cannot be normalised by min-max. Either the field is a time, and normalises, or it is ordinal, and needs a hand-assigned scale as the load field has. It was made a time in seconds, in keeping with the description in Chapter 4.

Chapter 6

Validation

6.1 Methodology

The validation proposed in this chapter should demonstrate that the discovery architecture proposed in Chapter 4 and implemented in Chapter 5 satisfies the rationales that motivated its development. Three rationales guided the design, and each can be phrased as a question the experiments must answer. Does the mechanism impose a low network overhead? Does it place a low computational demand on the devices that run it? And can its result be used natively by a standard industrial execution environment? The first two questions are quantitative and are answered by measurement. The third is demonstrative and it is answered by driving a real binding in a standards-compliant runtime.

The evaluation rests on the principle that every reported number must be a property of the mechanism itself. A measurement must not depend on the platform it is taken from, nor reflect an artefact of how the experiment was arranged. This principle decided both what is measured and what is deliberately left out, and it is worth making the second part explicit beforehand, because two quantities that might seem natural to report are excluded by it.

One aspect that is intentionally excluded is the *selectivity* of each phase, meaning the proportion of candidates removed during the pre-filtering and ranking stages. This value is not an inherent property of the mechanism itself, it is entirely dependent on the population created for the experiment. A scenario with a specific ratio of matching and non-matching providers will naturally produce a corresponding level of filtering. Reporting this proportion would therefore only confirm that the mechanism eliminates the candidates it was designed to eliminate.

Although the composition of a population determines how many candidates are ultimately discarded, it does not determine *when* they are discarded or how much information has already been exchanged before that happens. For example, a device that does not belong to the requested category may remain silent, resulting in no additional cost, or it may respond before being rejected, meaning that its response has already been received and the associated cost has already been computed. This difference is determined by the behaviour of the probing layer rather than by the

composition of the scenario. As a result, it is reflected in the results through the cost introduced by an additional device or a set of them.

The other exclusion is *raw discovery time*. The requester waits a fixed interval after probing before concluding that every response has arrived, and this interval dominates any wall-clock measurement by orders of magnitude. Reporting raw discovery time would report the chosen timeout constant rather than the mechanism. What is reported instead is the part of the elapsed time the mechanism actually controls, together with the arrival instants of the responses themselves, which the timeout truncates.

The final validation process must demonstrate that the proposed architecture can be deployed within an industrial context using standard industrial tools. This approach differs from a purely quantitative architecture evaluation, as it is demonstrative and requires a practical implementation of the architecture. This process is detailed in the final section of this chapter, which presents a conceptual CPPS design developed using the DINASORE framework in compliance with the IEC 61499 standard.

6.1.1 Structure of the Validation

The experiments are grouped into four categories, which are designed to answer the questions stated in the methodology:

1. **Correctness** ensures that the discovery mechanism behaves as intended. The two-phase discovery process identifies providers belonging to the requested category, filters and ranks them, and ultimately selects one that satisfies the required semantic constraints. This property can be expressed as a sequence of sets. Supposing that R denote the set of providers that respond to the initial probe, the pre-filter then constructs the viable subset $V = \{\, c \in R \mid \mathrm{hard}(c) \,\}$ by retaining only those candidates that satisfy all mandatory conditions. The ranking stage orders the elements of V without removing any of them, after which the second phase evaluates the ordered candidates until it finds the first provider c for which $\mathrm{match}(\mathrm{TD}(c))$ is asserted.

   Correctness should guarantee three conditions: only providers belonging to the requested category can appear in R and non-matching advertisements should be discarded by the mDNS layer. Any provider that fails a mandatory condition is excluded during the pre-filtering stage and cannot appear in V. Finally, the selected provider is guaranteed to satisfy the complete semantic evaluation.

2. **Performance** quantifies the cost of a single discovery and highlights attributes from both the architecture and chosen protocols:

   - **Network overhead** is measured by wrapping the requester's socket operations and counting the application payload exchanged with the providers, across both UDP, used by both systems for announcement and by the proposal for retrieval, and TCP, used by the baseline for retrieval. All protocol headers are excluded, since they are identical

for a given transport and cannot distinguish one architecture from another. Traffic generated by any device unrelated to the tests is excluded from the count.

- **Waiting against computation** is measured with two clocks bracketing the same work: wall-clock time, which advances during blocked waiting, and process Central Processing Unit (CPU) time, which advances only while the process executes. The difference between the two is the time spent waiting on the network. The process CPU time in isolation is the computation cost of the funnel.
- **Multicast intrinsic behavior** is the distribution of instants at which responses reach the requester, measured from the moment the probe leaves. It characterises the latency the mechanism is subject to, and is reported instead of raw discovery time.

3. **Scalability** characterises how the cost increases as the number of providers in a segment grows and analytically models the associated overhead, allowing the mechanism to be compared with a legacy baseline rather than relying on a single measured point. The population expands in two distinct ways that must be considered separately, as the architectures handle them differently: an increase in the number of devices providing the requested capability, and an increase in the number of devices on the same segment that provide other capabilities.

4. **Compliance** closes the validation with a practical demonstration that the result of a discovery can be consumed natively by a standard industrial execution environment. The standard chosen is IEC 61499, for two main reasons: it is a reference architecture for distributed industrial control and its environment was already explored to demonstrate plug and produce mechanisms via RESTful RTEs (Chapter 3), which is compliant with the proposed architecture.

6.2 Measurement and Qualification of the Architecture

The testbed observes the mechanism as a module and never modifies it. No timing call and no counter lives inside the discovery code, which means all instrumentation is wrapped around it. This is what allows the measurements to be attributed to the architecture rather than to the act of measuring it.

Because a single run is subject to jitter in the mDNS layer and the scheduler, each configuration is run repeatedly and the results are reported using the median across repetitions, with variability shown as the range between the 10th and 90th percentiles. The median is used instead of the mean because occasional resolver retransmissions produce samples that are an order of magnitude higher than the others. These samples represent real events and are therefore not removed, but they should not disproportionately influence the reported central value. For measurements that remain constant across repetitions, such as byte counts, the median equals that fixed value and the percentile range collapses to a single point. The first repetition of each run is discarded because it reflects interpreter warm-up effects and is typically two to three times slower than the subsequent repetitions.

6.2.1 Properties of the Environment Affecting Measurement

Three properties of the environment shape how the data must be collected, and all three must be understood before the results, since the testbed is constructed specifically to account for them. The first two are properties of the design. The third one was discovered empirically while building the testbed, and forced a change in its physical layout.

The first is the *fixed probe timeout*. Because the requester cannot know how many providers will answer, it waits a fixed interval and then proceeds. This interval is always paid in full, so it dominates any measurement of elapsed time and would mask the quantity of interest. The measurements therefore separate the computation cost from the waiting, and use the recorded arrival instants to reason about latency independently of the constant.

The second is the *local mDNS cache*. The mDNS stack caches recent responses, so a second probe issued shortly after a first may be answered from memory and appear almost instantaneous, measuring the cache rather than the network. Each measured discovery therefore starts from a fresh resolver with a cold cache, so that every sample reflects a genuine network exchange.

The third is *response batching between co-located providers*. During the construction of the testbed, providers hosted on a single machine were observed to answer a probe in a single burst. A packet capture showed all responses leaving the shared host within the same microsecond, regardless of how many providers are hosted. The cause is that the responses never cross a physical link separately, so the responder that governs a real provider population collapses into one transmission. The testbed therefore runs every provider as an independent host on the network, each with its own address and its own mDNS responder.

6.2.2 A Descriptive Model of the Overhead

The scalability analysis models the traffic generated by a discovery as the sum of two components: probe responses and the metadata documents retrieved,

$$C = \beta_1 |R| + \beta_2 k \tag{6.1}$$

where $|R|$ is the number of providers that respond to the probe, k is the number of retrieved metadata documents, and β_1 and β_2 represent the measured cost of a single response and a single document, respectively. Rather than a predictive model, this expression summarizes the measurements obtained for the evaluated platform and workload.

The key difference between the two architectures lies in the number of probe responses. Let P be the number of devices on the network segment and ρ the fraction that belongs to the requested service category. If devices respond only to requests matching their service type, then $|R| = \rho P$. If every device responds to every probe, then $|R| = P$. In both cases, the same model applies; only the first term changes according to the response strategy, which is examined experimentally through the composition analysis presented next.

6.3 Testbed Design

The testbed follows from the preceding two sections: it must exercise the four categories of Section 6.1.1 while accommodating the three properties of Section 6.2.1. Two of them fix its physical layout directly. Because arrival behaviour is meaningless with co-located responders, every provider runs in its own container with its own address on the physical network, so that each is an independent mDNS host. Because byte counts must reflect the mechanism rather than the host, the requester runs alone in its own process on a separate machine, reaching the providers over the real network.

The evaluation uses a single harness configured along three independent axes rather than a set of separate testbeds. Keeping the axes in one harness is what allows them to be crossed, so that the behaviour of the mechanism can be observed as more than one factor varies at a time.

The **density axis** varies the number of providers on the segment, and is the base for the scalability evaluation. The **composition axis** varies how that population is divided between the requested top-level category and the remaining ones, holding the total fixed. The **match position** defines the index at which the compatible device sits in the ranking, and therefore how many CoAP retrievals the measurement runs before reaching the final candidate. The two systems are never active simultaneously, since their populations would share the segment and each would measure the other's traffic as interference.

6.3.1 Execution Environment

The providers are hosted on a Raspberry Pi 3 Model B, with a quad-core (ARMv7) 64-bit Broadcom BCM2837 processor and 1 GB of memory, connected to the segment by 100BASE-T Ethernet over a wired link. The board runs a minimal Linux Debian installation with Python 3.13 and Docker Engine 29.6.2. The provider farm is defined once and holds 32 virtualized devices, sixteen of the welding category (used as the request target by all the experiments) and sixteen distributed over four other top-level categories, from which each experimental cell brings up the subset it needs. No cell exceeds sixteen simultaneous providers, which is the largest population the campaign measures. Each provider runs in a container built from the `python:3.12-slim` image, so that the runtime of the mechanism is identical in every container and independent of what the host provides.

The selection of this specific single-board computer is justified by its alignment with the constrained edge devices targeted by the proposed architecture. Consequently, hosting the population on this platform serves to validate whether the mechanism is lightweight enough to concurrently run multiple instances. To further ensure this choice remains representative, the processor clock frequency was capped at 600 MHz. Figure 6.1 illustrates the physical deployment of the testbed environment.

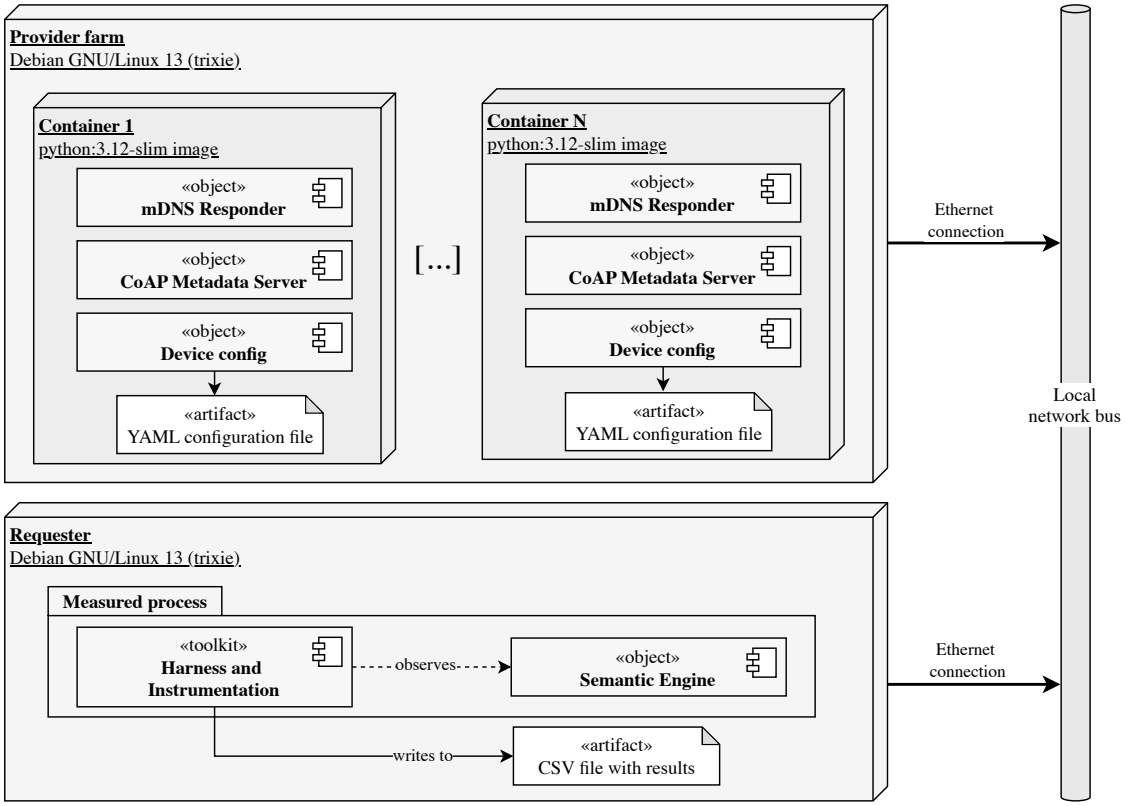


Figure 6.1: Testbed physical deployment.

The requester runs on a separate machine within the same network segment, with its execution environment hosted inside a Virtual Machine (VM). Although the physical platforms differ, this VM was carefully configured to approximate the resource constraints of the edge device serving the mDNS and CoAP responders, ensuring a comparable baseline. Developed via VMware Fusion using the provider farm's Linux distribution, the VM was capped at 1 GB of memory and shared only a single core of the host's 1.6 GHz dual-core Intel Core i5 (5250U) processor to mimic the limited processing power.

6.3.2 The Reference Baseline

To provide context for the results derived from the validation process, a baseline that represents the main method in industrial SOA for distributed discovery is measured under the same conditions. This uses WS-Discovery for announcement and SOAP-over-HTTP for metadata retrieval, the approach examined by Dai et al. [12, 13]. The baseline acts as a reference point for interpreting the figures of the proposed mechanism.

The choice of this particular stack converges with the proposed architecture in the sense it is an ad-hoc, fully distributed discovery with no central registry, and it decomposes into the same two phases, a multicast announcement stage followed by a per-candidate metadata retrieval. This structural similarity is itself a reason for the choice because it makes the comparison interpretable. In other words, the two systems answer the same question through the same shape of exchange,

so the measured difference isolates the weight of the performance rather than a difference of architecture.

A reference is only useful if it is the strongest reasonable version of the approach. Three decisions enforce this:

1. The baseline carries semantics in WS-Discovery Scopes, the mechanism the specification itself provides for discovery-time metadata, which is the natural counterpart of the proposal's TXT record and matchmaking.

2. It is a genuine two-phase funnel: a probe filters by service type at the responder and the Scopes carried in each Probe Match are evaluated by the requester, mirroring the service name and the TXT record of the proposal, then an HTTP retrieval fetches a WSDL document for each survivor.

3. The WSDL document carries the same semantic content as the proposal's Thing Description, so that a byte comparison measures the difference in serialisation and transport, not a difference in information.

Both systems are exercised against the identical population of devices, generated once per random seed and translated into each system's native form, so that any difference reflects the two architectures rather than two scenarios. One asymmetry is retained on purpose: the proposal treats the device zone as a ranking criterion, while WS-Discovery has no native ranking and can only filter, so the baseline filters on the same mandatory condition the proposal treats as mandatory and cannot rank

6.4 Binding Inside an IEC 61499 Runtime

The measurement mechanism that evaluated the final process of a service discover has to answer a different question, the one Section 6.1.1 states it as demonstrative. This answer completes the validation by linking the findings back to the central questions of this dissertation.

6.4.1 The RTE

The chosen environment is DINASORE [42], an IEC 61499-compliant runtime. While the mechanism was encapsulated into three SIFBs as detailed in Section 4.5, minor modifications to the DINASORE runtime were required. These adjustments were necessary to restore compatibility with Python 3.11 and the updated deployment payload format enforced by the 4DIAC IDE. Aside from resolving these integration defects (summarised in Table 6.1), no architectural changes were made which means the applications are booted and driven through the standard management interface used by the 4DIAC IDE, version 3.2.1.

#	Affected file	Issue & Solution
1	`configuration.py`	Replaced `inspect.getargspec` (removed in Python 3.11) to prevent block creation failures.
2	`fb_resources.py`	Stripped library namespaces from type names (e.g., `iec61499::system::`) to match DINASORE's file system lookup. This compatibility issue was introduced on newer releases of the 4DIAC IDE.
3	`fb_interface.py`	Removed the obsolete `time` attribute from event watches to prevent IDE read aborts.
4	`fb_interface.py`	Formatted monitored `STRING` watch values as compliant IEC 61131-3 literal types used by the IDE.

Table 6.1: Runtime compatibility fixes applied to DINASORE for execution with recent Python versions and interoperability with 4DIAC.

6.4.2 The Application

The developed application involved creating new FB and SIFB types and descriptions that were loaded both on the DINASORE repository and the 4DIAC external library. Figure 6.2 illustrates the application design for the service provider machine `weldbot_07`, and Figure 6.3 illustrates the service consumer side.

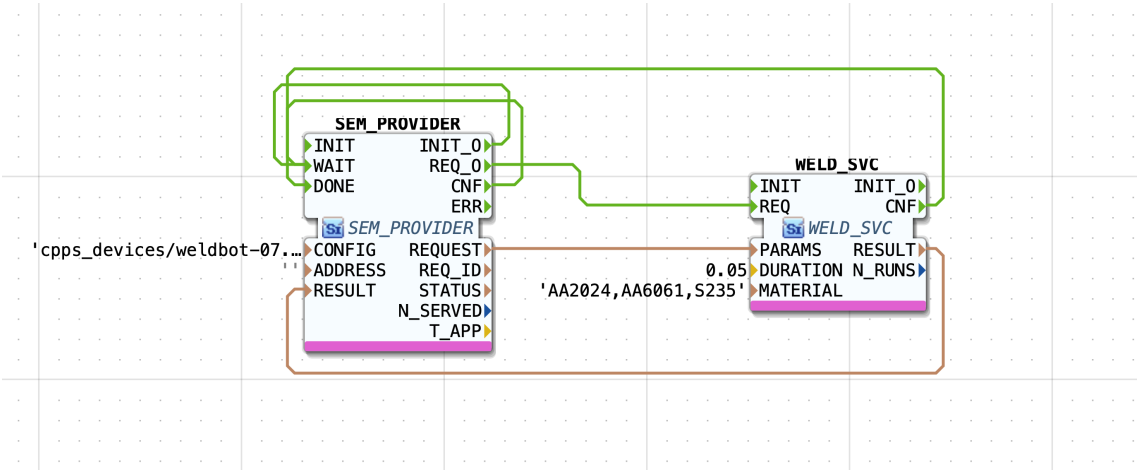


Figure 6.2: Provider side FB system diagram.

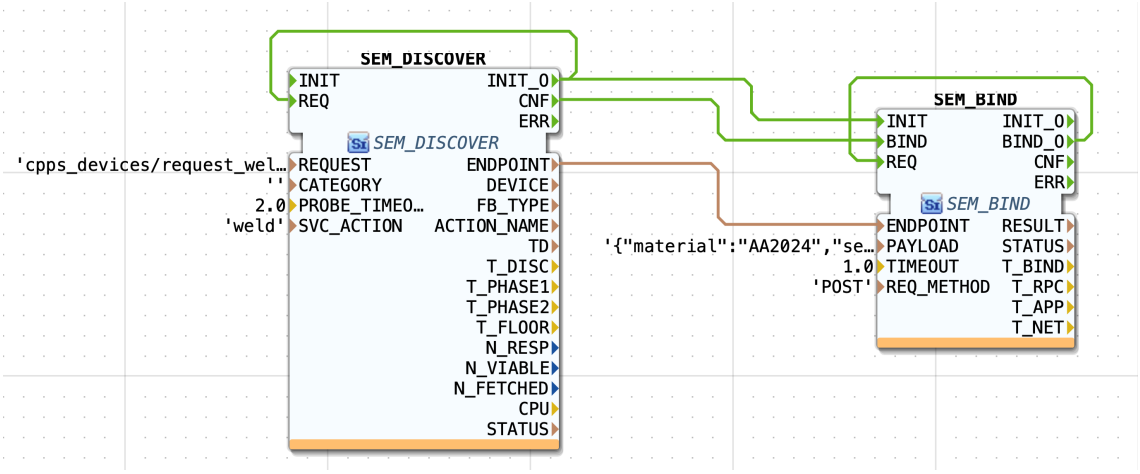


Figure 6.3: Consumer side FB system diagram.

On the consumer side, `SEM_DISCOVER` executes the discovery funnel defined by the proposed architecture and presented in Chapter 4. Its outputs provide the endpoint of the selected service, the FB type that implements it on the provider, and the corresponding TD. All the remaining outputs are measurement and verification used to collect the discovery metrics to be reported. `SEM_BIND` then uses the discovered endpoint to establish a runtime binding. It intentionally separates the binding process into two events: `BIND` establishes the connection accordingly to the interface exposed by the provider inside the TD, while `REQ` invokes the service. This distinction also makes it possible to measure the initial discovery and connection overhead independently from the cost of subsequent service invocations.

On the provider side, `SEM_PROVIDER` advertises the device under its top-level category, serves the TD at the path specified in its TXT record, and translates each request received at the CoAP endpoint into an event within the local application. A response is only sent after the application has completed its execution and produced the corresponding result. The capability exposed as the service is implemented by `WELD_SVC`, a standard Basic Function Block (BFB) containing an input event, an algorithm, and an output event. The function block itself has no knowledge of discovery, CoAP, or the TD document, and blocks execution for a fixed 50 ms on every invocation. Fixing this duration is what allows the time spent inside the provider's application to be separated from the time spent reaching it.

A service invocation follows the sequence illustrated in Figure 6.4, from the request event to the response returned to the client.

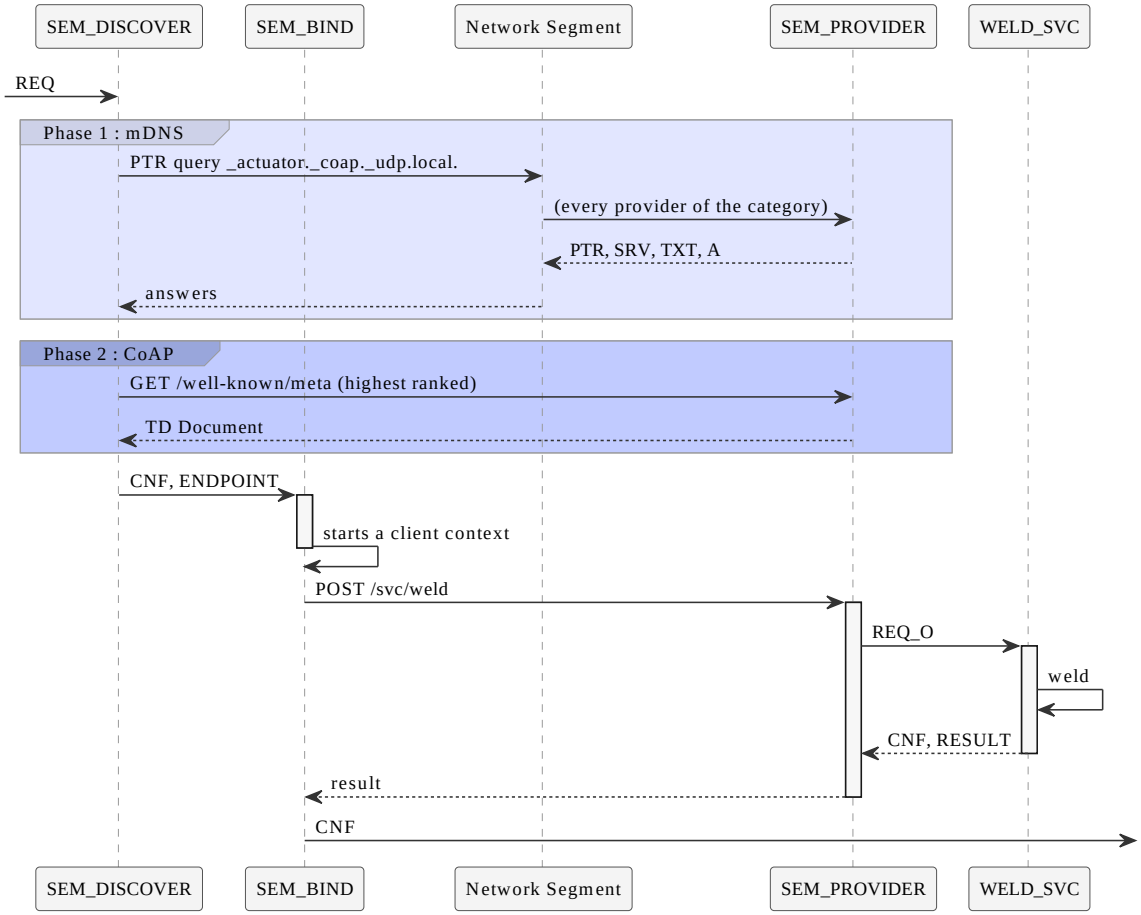


Figure 6.4: Sequence of one service invocation, from the request event raised by the consumer application to the result returned by the provider's FB network.

Following the provider application in the order events are processed clarifies the execution flow. `SEM_PROVIDER.WAIT` remains idle until a request arrives at the CoAP endpoint. The `REQ_O` event forwards the request to the application together with its payload on `REQUEST`, `WELD_SVC` performs the requested operation, `CNF` is passed to `DONE`, and only then is the response transmitted back to the client. The consumer application mirrors this process in reverse. Components downstream of `SEM_DISCOVER` remain unaware of which specific device provides the service, since the endpoint is supplied through a regular data connection like any other runtime value.

The device identifiers used throughout the validation, including the model designation `model_v3`, are fictitious. They stand for the fields a real catalogue would carry and were chosen so that no claim in this dissertation depends on the characteristics of any industrial product.

6.4.3 Instantiating the Discovered Block at Runtime

In the application described in Section 6.4.2 the proxy block which calls the remote service (`SEM_BIND`) is drawn in advance and it is only the endpoint that is determined at runtime. This

solution has already removed the device mapping from the engineering tool, but it still necessitates the consumer application being drawn against a known block type.

A second variant removes this assumption as well. In this version, a third SIFB, `SEM_DEPLOY`, replaces `SEM_BIND` on the consumer side. Instead of binding an existing function block, it reads the FB type from the `cpps` field of the retrieved TD, creates a corresponding instance in the running configuration using the IEC 61499 management commands, connects it to the function blocks that consume its output, and binds it to the discovered endpoint without interrupting the application. Figure 6.5 ilustrates the connection diagram on 4DIAC IDE.

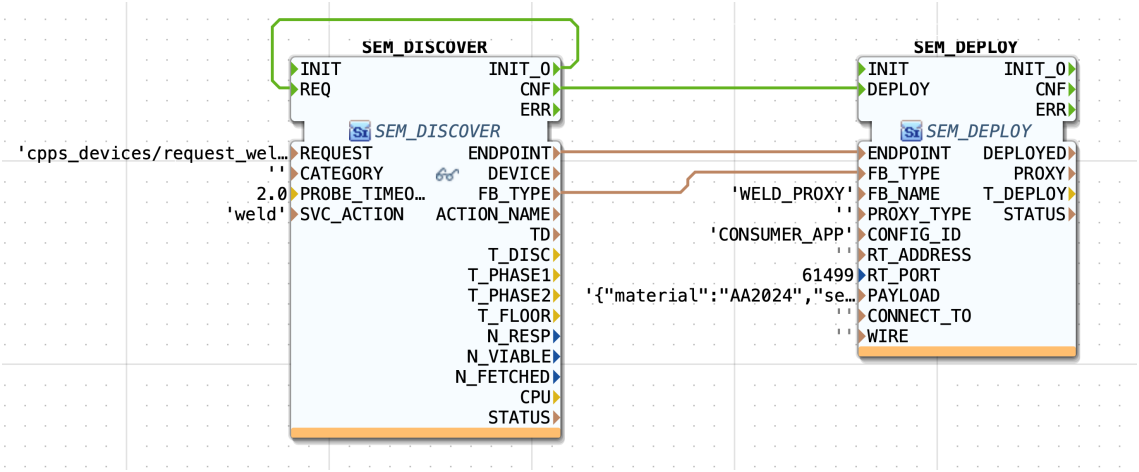


Figure 6.5: Semantic Deployer FB integrated with the Semantic Discover.

The `ENDPOINT` carries the address the discovery resolved and is written into the created instance. `CONNECT_TO` and `WIRE` carry the event and data connections between the proxy and the blocks that consume its output, that is, the connections an engineer would otherwise have drawn, expressed as source and destination pairs. `CONFIG_ID`, together with `RT_ADDRESS` and `RT_PORT`, identifies the configuration and the management interface the commands are addressed to. Finally `FB_NAME` determines identity, and identity determines cost. The commands that create the instance and wire it are issued only the first time a given name is deployed, so a later discovery that resolves to a different endpoint reuses the existing instance and rewrites only its endpoint before triggering the binding again.

The block also reports the deployment on `T_DEPLOY`, which measures the exchange with the management interface alone, while the binding and the invocation are measured by the instance it creates, on that block's own `T_BIND`, `T_RPC`, `T_APP` and `T_NET` outputs. Further illustration containing the detailed algorithm implemented by the block is described in Figure 6.6

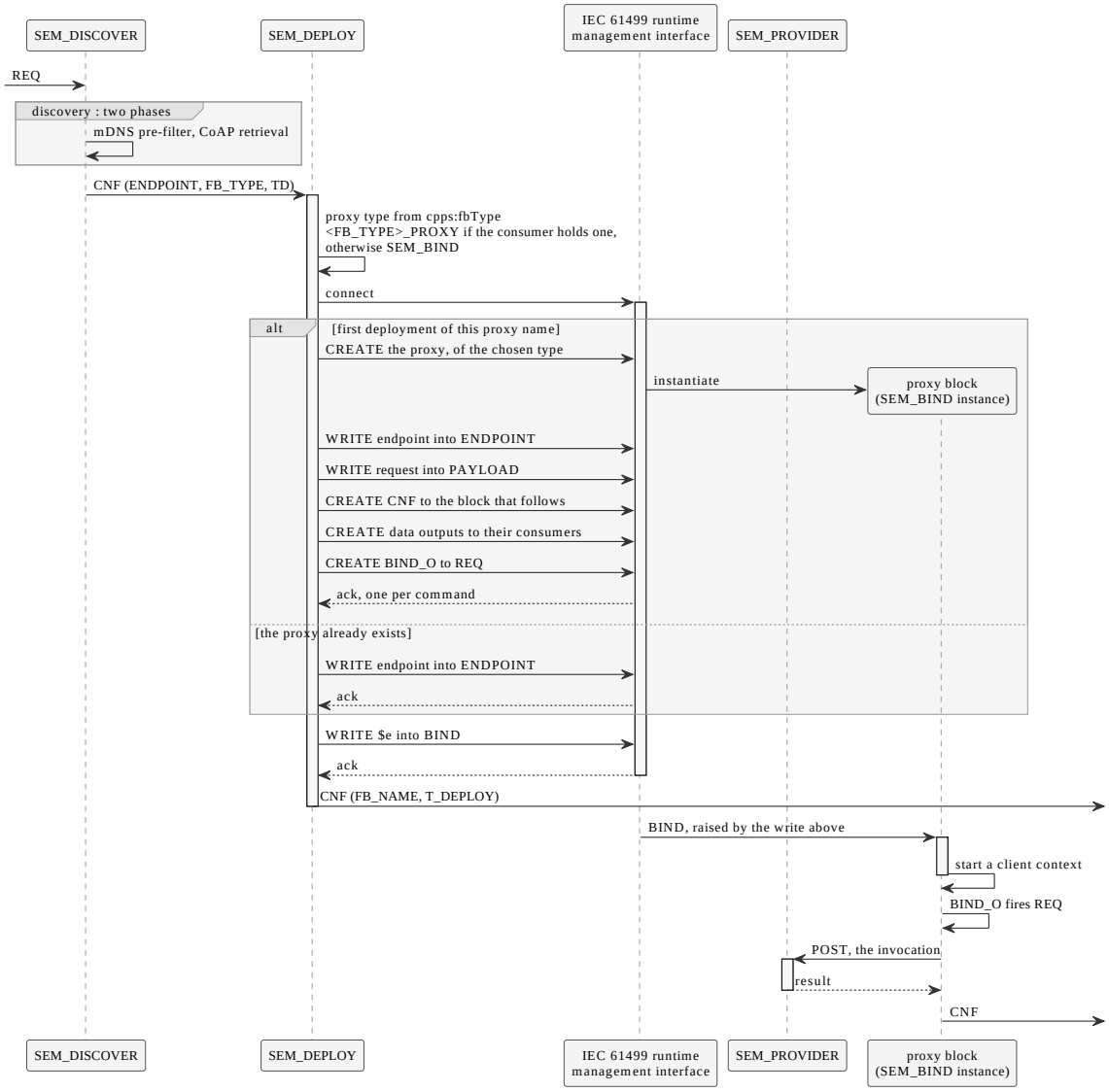


Figure 6.6: Semantic Deployer Sequence Diagram.

This variant is included in the validation because it distinguishes between two capabilities that are often considered together. The first is that a discovery result can configure an already existing function block, as demonstrated by the main application. The second is that the function block itself does not need to exist at design time, which is demonstrated only by this variant and represents the stronger form of runtime binding addressed by the second research question of this dissertation.

Chapter 7

Results

The measurements collected during the campaign follow the sequence described in Section 6.1.1. Before conducting the experiments, the network segment was characterized to establish a reliable baseline for interpreting the subsequent results. The chapter concludes with the compliance demonstration, which is the only experiment that evaluates the mechanism end-to-end in a standard industrial runtime environment.

7.1 Prior Network Segment Characterisation

All the measurements reported in this section include byte counts, packet counts, and timing intervals on the order of a few milliseconds. Any network segment that was already congested or carrying multicast traffic similar to that generated by the proposed mechanism would contribute to these measurements, making it impossible to distinguish pre-existing traffic from traffic introduced during the experiments. To account for this, the network segment was characterised during a baseline period of 60 seconds immediately before the measurement campaign. Monitoring was performed simultaneously at both endpoints using an instrument deployed on the requester and provider hosts which includes the tools *iperf3* and *tshark* and the Python library *psutil*. The collected data included interface counters sampled once per second, protocol-classified packet captures, and an active throughput test between the two hosts in order to confirm the available network bandwidth. The 60 seconds observation period was divided into three phases: 30 seconds of background traffic monitoring, followed by 15 seconds of TCP load generation and 15 seconds of UDP load generation. Table 7.1 summarises the result.

Quantity	Measurement endpoint	Value
Traffic present on the segment (60 s capture)		
Total	requester	122 616 pkt / 23.26 Mbit/s
Multicast	requester	13 pkt / 4 074 B
Broadcast	requester	142 pkt / 5 964 B
mDNS (port 5353)	requester	14 pkt / 4 546 B
WS-Discovery (port 3702)	requester	0 pkt
CoAP (ports 5683/5684)	requester	0 pkt
Capacity of the path (iperf3)		
TCP throughput	both	79.75 Mbit/s
TCP retransmissions	provider	0
UDP offered rate	provider	10.01 Mbit/s, 11 232 datagrams
UDP received rate	requester	10.30 Mbit/s (median)
Interface health, summed over the three windows		
Receive drops	requester	0
Receive errors	requester	0
Receive drops	provider	30
Receive errors	provider	0
Transmit drops and errors	both	0

Table 7.1: The segment, characterised over the 60 seconds immediately before the campaign.

Three observations provide context for the measurements that follow. First, the network segment carried almost no multicast traffic: thirteen multicast datagrams and fourteen mDNS packets over the whole minute.[1] No WS-Discovery or CoAP traffic was observed, confirming that no responders or requesters were active on the network. The measurements therefore reflect a network segment where the proposed mechanism was effectively the only source of the relevant traffic.

Second, the link capacity was not a limiting factor. The network sustained 79.75 Mbit/s of TCP traffic without retransmissions, and a 10 Mbit/s UDP stream was received at the expected rate, with no packet drops or receive errors at either endpoint. Since a complete discovery exchanges less than one kilobyte of data, none of the results reported in this chapter are constrained by link capacity, and the measured intervals should not be interpreted as transmission delays.

Finally, the provider host recorded 30 receive drops during the measurement period, distributed evenly across the three observation windows and never exceeding one drop per sample. No receive errors were observed. These drops are consistent with occasional frame loss caused by the limited processing capacity of the 600 MHz platform used in the experiments. Although too few to explain any of the measured effects, they are reported here for completeness.

[1] The two counters are not nested. mDNS is classified by port and multicast by destination address, so a unicast mDNS response is counted in one and not in the other, and the two totals need not agree.

7.2 Correctness

Correctness was evaluated using the populations created for the scalability campaign, allowing the sets defined in Section 6.1.1 to be measured empirically. Here, R denotes the set of providers that responded to the probe, V the subset that passed the pre-filter, and k is the number of metadata documents retrieved by the second phase before terminating. A population of sixteen devices was constructed using five different compositions, ranging from a single device of the target top-level category against fifteen devices from the remaining ones (Human-Machine Interface (HMI), sensor and gateway), to a population consisting entirely of sixteen devices from the target top-level category. At the last run, half of the devices is set to be viable after the first phase evaluation. Each composition was executed forty times on each architecture using two interleaved passes. The results are summarised in Table 7.2, while Figure 7.1 illustrates the funnel behaviour produced by the two extreme compositions.

Architecture	Composition	$\lvert R\rvert$	$\lvert V\rvert$	k	Resolved
Proposed	1 of 16	1	1	1	40 of 40
	2 of 16	2	1	1	40 of 40
	4 of 16	4	1	1	40 of 40
	8 of 16	8	1	1	40 of 40
	16 of 16	16	8	1	40 of 40
Baseline	1 of 16	1	1	1	40 of 40
	2 of 16	2	1	1	40 of 40
	4 of 16	4	1	1	40 of 40
	8 of 16	8	1	1	40 of 40
	16 of 16	15-16	7-8	1	40 of 40

Table 7.2: Cardinality of the three sets across forty repetitions of each composition. A single value means the same number was observed in every repetition while a range means variation.

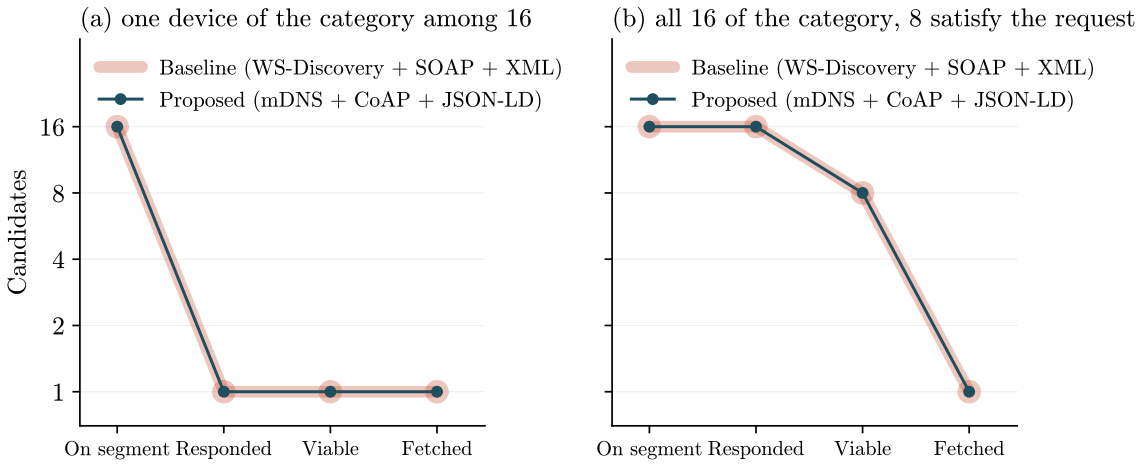


Figure 7.1: The funnel over the same population of sixteen devices, in the two extreme compositions.

The first condition holds without exception meaning no provider from a different category entered the R set. When a single welding robot was deployed alongside fifteen devices from other categories, the answer set contained exactly one element in each of the forty repetitions. The same behaviour was observed for every other population composition, with $|R| = r$ and no variation between runs. A responder that does not recognise its own service type in the request simply does not reply, so the fifteen devices from other categories were not filtered out because they simply never responded to the probe. The same holds for the baseline under a conformant responder.

The second condition also holds meaning that in populations containing sixteen devices of the requested category, where eight were constructed to satisfy the hard conditions (mDNS metadata), $|V|$ was consistently eight across all repetitions. When only a single device of the requested category was present, $|V|$ was always one. The pre-filter removed only the candidates that failed a mandatory condition while retaining those that satisfied all requirements, exactly as intended by the metadata included in the service announcement.

Finally, the third condition is also satisfied, although it requires an important qualification. In every sample of the experimental campaign, the second phase retrieved exactly one document under both architectures, a TD for the proposed architecture and a WSDL document for the baseline. This indicates that the ranking consistently placed a provider that passed the full semantic evaluation in first position. That outcome is expected for the populations used here, where the ranking criterion aligns with the mandatory conditions, but it should not be interpreted as a general property of the mechanism. In a population where the highest-ranked candidate fails a condition expressed only in the metadata document, the requester would need to retrieve more than one document ($k > 1$). The effectiveness and the cost of that scenario is analysed separately in Section 7.3.2.2.

Every discovery attempt resolved on both architectures, 200 of 200 for the proposed mechanism and 200 of 200 for the baseline. The only variation in set size appears in the last composition, where sixteen providers answer at once. The baseline answer set ranged from fifteen to sixteen and the viable subset from seven to eight, indicating that a reply occasionally failed to reach the requester within the response window. No repetition failed to return a result.

The precise cause was not isolated experimentally, but two explanations are consistent with the observations. Sixteen responses of approximately 2.5 kB, transmitted within a bounded delay window, contend for the shared network segment more than sixteen responses of roughly a quarter of a kilobyte. And the repeated queries of the proposed mechanism carry the answers already received, so providers whose records are listed stay silent and repetition does not multiply the traffic on the segment, for which the baseline protocol has no equivalent facility.

7.3 Performance

7.3.1 Network Overhead

This first measurement establishes what a complete discovery costs at the physical link level. The scenario is the simplest the mechanism admits: a single provider on the segment, satisfying the

request, so that the funnel resolves with exactly one metadata retrieval and the two phases separate cleanly. Every figure in Table 7.3 was identical in every repetition, so the standard deviation of each is zero.

Exchange	Application Payload		Wire	
	Sent	**Received**	**Sent**	**Received**
mDNS question + response	44	239	86	281
CoAP GET + Thing Description	22	523	64	565
Effective discovery ($k = 1$)	**66**	**762**	**150**	**846**

Table 7.3: Cost of the exchanges, in bytes, a discovery requires for a single provider, all with zero variance.

Each phase is given both as application payload and as it appears on the wire, where every datagram carries a further 42 bytes of Ethernet, IP and UDP framing.

Within the two-second probe window, the implementation emitted seven datagrams rather than the two the discovery required: three questions in each of the protocol's two response modes, plus the metadata request. The gross cost of the discovery was therefore 364 B of payload sent against 66 effective, and 658 B on the wire against 150. The received side was unaffected, at 762 B of payload and 846 on the wire in both readings, since the provider answered once. The difference between the two readings is the price of the protocol's reliability strategy, and it is proportional to the probe window and the defined timeout.

A second run, under same conditions and against the same machine, was executed for the WS-Discovery baseline model. The readings are put side by side in Table 7.4:

Architecture	Effective		Gross	
	Payload	**Wire**	**Payload**	**Wire**
Proposed (mDNS+CoAP)	828	996	1126	1504
Baseline (WS-Discovery+SOAP)	4170	5042	7028	8068

Table 7.4: One complete discovery of the same service by the two systems, in bytes, for a single provider. The *effective* figures count only the packets required by the discovery protocol, whereas the *gross* figures include all traffic emitted during the probe window.

It is important to note that the measurement of the exchange executed by the baseline model also accounts the TCP handshake process introduced by its connection-oriented architecture. Additionally, the gross reading has the same meaning for both systems. Just as the proposed mechanism repeats its question three times in each of the protocol's two response modes, the baseline emits four probes where one would have sufficed, and the provider answers twice, so the gross figures include three probes and one response that the discovery did not need. The difference between

the two readings is again the price of a reliability strategy over an unacknowledged transport, and again proportional to the window rather than to the mechanism.

7.3.2 Waiting Against Computation

Figure 7.2 reports the time decomposition for both systems, over a provider population from 1 to 16. All values in this section are medians over 40 samples per cell, collected in two interleaved passes and the first repetition of each run is discarded, since it may carry the interpreter warm-up cost.

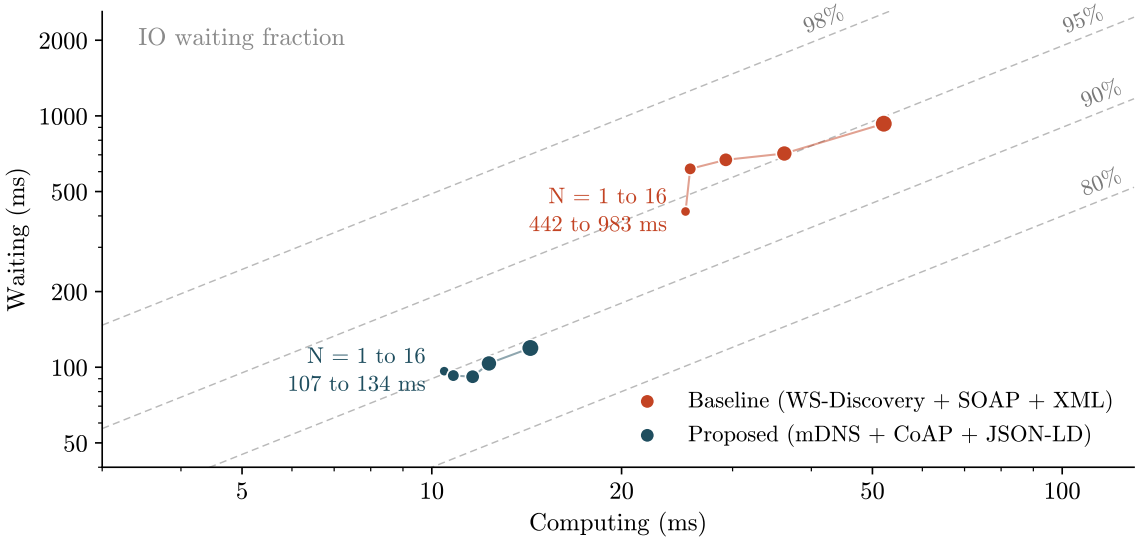


Figure 7.2: Waiting against computation fraction between the proposed architecture and the baseline. Considering total discovery time.

Both systems spend most of the discovery time waiting: the proposed mechanism 89 to 90%, the baseline 94 to 96%. Each fraction is normalised by the total time of the respective architecture and should not be read alone. The IO fraction rises whenever waiting grows faster than computation, which is what a slower mechanism does. At 16 providers the baseline waits 7.8 times more than the proposed mechanism and also computes 3.6 times more. The totals are 134 ms against 985 ms.

The total time spent both on computation and IO waiting can be separated into two phases. The first is the network discovery, measured from the moment the probe is sent until the last response is received. The second is where the semantic documents are fetched and processed. It is important to note that the two phases wait for different reasons and scale differently.

7.3.2.1 Phase 1: Network Discovery

The parameter varied is the number of providers that answer the probe: 1, 2, 4, 8 and 16. The number of documents fetched in the second phase is held at one, so that any change comes from the first phase alone. The results are shown in Figure 7.3.

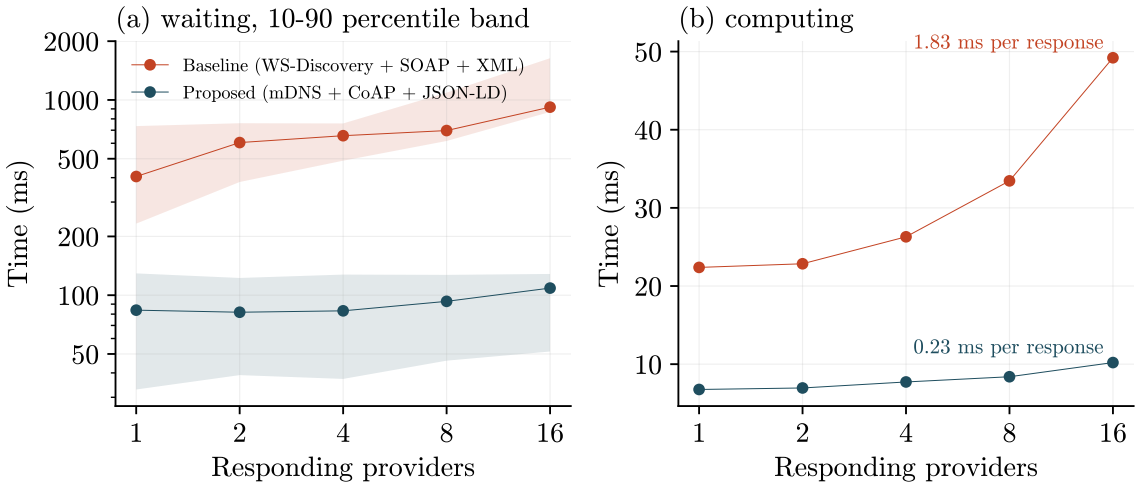


Figure 7.3: Phase 1 waiting and computing against the number of responding providers.

Waiting does not increase substantially with the population in the proposed mechanism. It rises from 84 ms with one provider to 109 ms with sixteen, corresponding to a slope of 1.8 ms per responder, while the dispersion remains close to 28.9 ms across all population sizes. Both protocols require a responder to delay its answer by a randomly chosen interval, so in both cases the last arrival is the maximum of N independent draws. What differs is the width of the interval each standard mandates. RFC 6762 requires an mDNS responder answering a shared-record query to wait a uniformly random 20 to 120 ms [9], a window only 100 ms wide, and the measured dispersion of 28.9 ms is what a uniform draw over that window would give. Because the maximum of N draws saturates towards the upper end of a bounded window, adding responders moves the last arrival very little once the population is more than a few devices.

The baseline exhibits the same mechanism over a much wider window. WS-Discovery bounds each responder's independently at 500 ms [54], five times the mDNS interval, and waiting accordingly rises from 405 ms to 919 ms, a slope of 28 ms per responder. The values measured at the larger populations exceed the 500 ms bound itself, which indicates that queueing and per-response processing in the baseline stack contribute beyond the delay.

On the other hand, computation time increases in both mechanisms, but at different rates: 0.23 ms per response in the proposed mechanism compared with 1.83 ms in the baseline. This difference reflects the amount of processing required for each response on both architectures. The proposed mechanism only compares keys and values stored in a TXT record, whereas the baseline must parse an entire SOAP envelope.

7.3.2.2 Phase 2: Document Fetch and Semantic Evaluation

The parameter varied is k, the number of documents the requester fetches before it finds a match: 1, 2, 4 and 8. The number of responding providers is held at eight and the remaining ones are configured not to match the top-level category of the mDNS probe. The population is built so that the true match sits at a chosen position in the ranking, which makes k an input. Figure 7.4

illustrates the comparisons for both IO waiting time and computation time for phase 2 between each model.

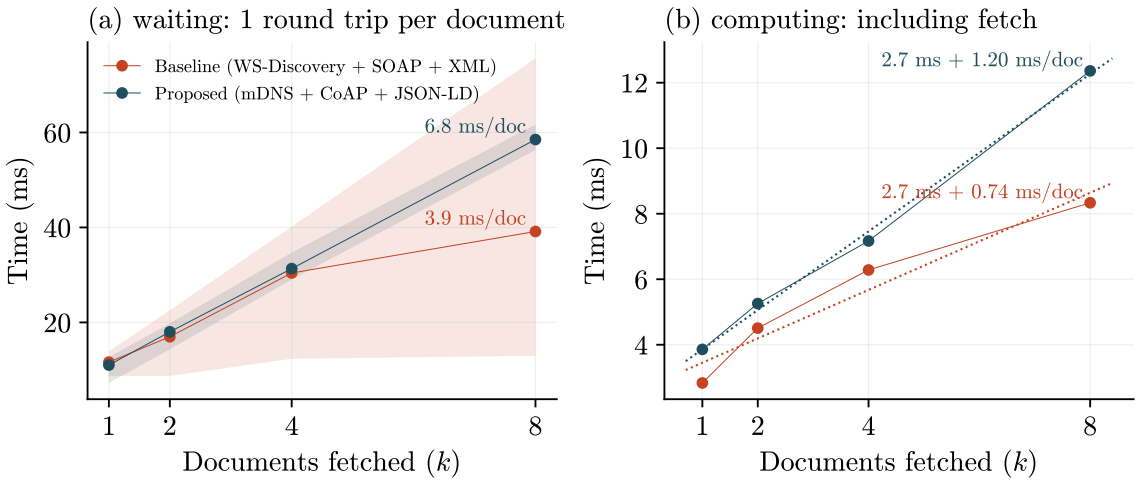


Figure 7.4: Phase 2 waiting and computing against the number of documents fetched.

Waiting is linear in k and passes through the origin within network noise. Each document costs one round trip, and the main difference between the two architectures is the transport protocol, which demands one TCP connection per candidate for the baseline. Computation is also linear, but has an intrinsic offset for both architectures, which are 2.66 ms for the proposed mechanism and 2.71 ms for the baseline and may assigned to the harness startup. The per-document cost is therefore constant, at 1.20 and 0.74 ms.

The per-document figures above put the baseline below the proposed mechanism, which would suggest that evaluating a SOAP document is cheaper than evaluating a JSON-LD one. The reason for this result is that the counter covers the whole second phase window, and what dominates it is the CPU of the client library performing the request, not the evaluation of the document. As illustrated in Figure 7.5 (a), a second run characterises the matching mechanism on its own, with the request stack excluded from the count.

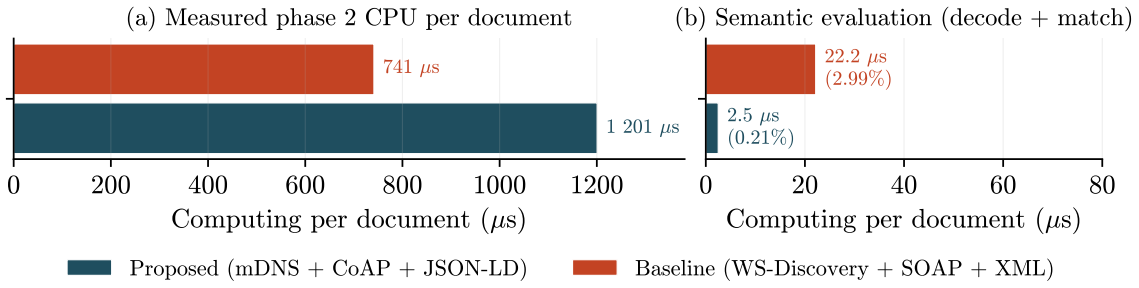


Figure 7.5: Cost of evaluating one document, measured in isolation, against the per-document CPU measured during the campaign.

Parsing and evaluating a semantic document requires only 2.5 µs in the proposed mechanism, compared with 22.2 µs in the baseline, making the XML-based implementation approximately 8.9

times more computationally expensive than the JSON-LD-based approach. As semantic evaluation accounts for only a small fraction of the measured phase-2 CPU time, the remaining processing cost is therefore attributable primarily to the transport stack and protocol implementation of each library.

7.4 Scalability

This section examines the additional cost of each extra device added to the network segment, the key factor in determining whether a mechanism can scale beyond a laboratory setting. The campaign keeps the total population fixed at sixteen devices while varying the composition of: r devices of the requested category and $16 - r$ devices from other top-level categories, with r taking the values 1, 2, 4, 8, and 16. Each configuration was measured twice, once with devices from other categories present and once without them, ensuring that the two cases differ only in the presence of those devices. Figure 7.6 presents the results for both conditions.

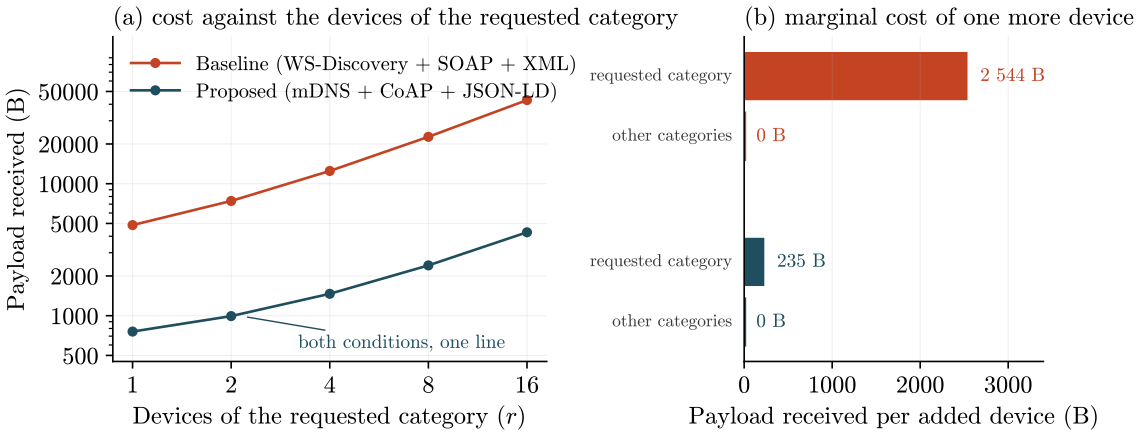


Figure 7.6: Payload received and marginal cost against the number of devices of the requested category.

For the proposed mechanism, both conditions produced identical results, matching exactly at the byte level. A request handled by a single welding robot generated 758 B of received traffic, regardless of whether the robot was alone on the network segment or accompanied by fifteen devices from other categories. This pattern remained consistent across every network composition that was evaluated.

In all four comparisons, the difference was exactly zero. The byte counts remained constant across all forty repetitions of every experimental condition. Devices outside the requested category imposed no communication overhead on the requester. Adding such devices did not affect the amount of received traffic, the number of packets exchanged, or the processing effort required to handle them.

For the baseline the two conditions are similar. In the condition where only the r devices of the requested category are present, which is what a conformant responder-side filter produces on a

full segment, the received payload rises in proportion to r, from 4858 B at r equal to one to 43014 B at r equal to sixteen. In the condition where fifteen devices of other categories are also present, the implementation received exactly the same amount of traffic.

	Proposed (mDNS + CoAP)				Baseline (WS-Discovery + SOAP)			
r	$\|R\|$	Payload	Pkts	CPU	$\|R\|$	Payload	Pkts	CPU
1	1	758	7	7.3	1	4858	8	26.7
2	2	993	8	6.8	2	7402	10	28.7
4	4	1 465	10	7.7	4	12492	14	31.1
8	8	2 405	14	8.5	8	22666	22	37.4
16	16	4 285	22	10.2	16	43014	38	55.4

Table 7.5: A population of sixteen devices, of which r belong to the requested category. Payload in bytes received, packets received, and first-phase CPU in milliseconds, as medians.

The measured marginal costs provide a concise summary of the results, as shown in Table 7.5. Each additional device offering the requested service increases the received traffic by 235 B in the proposed mechanism and by 2544 B in the baseline and, for devices outside the requested category neither mechanism incurs additional traffic.

Moreover, the CPU time spent on the first-phase is significantly lower for the proposed architecture. Analytically comparing with the baseline, it can be approximated by $6.5 + 0.23\,|R|$ ms, against $24.1 + 1.90\,|R|$ ms, indicating a processing cost of 0.23 ms and 1.90 ms per response, respectively. With sixteen devices on the network and only one matching the requested service, the proposed mechanism spends 7.3 ms in the discovery phase, while the baseline requires 26.7 ms.

7.4.1 The Overhead Model Instantiated

The discovery traffic can be described as the sum of two components: probe responses and the metadata documents retrieved,

$$C = \beta_1\,|R| + \beta_2\,k \tag{7.1}$$

where $|R|$ is the number of providers that respond to the probe, k is the number of retrieved metadata documents, and β_1 and β_2 are the measured costs of a single response and a single document, respectively. This expression is an empirical summary of the measurements collected for the evaluated platform rather than a general predictive model.

The difference between the two architectures is the number of probe responses. Let P be the number of devices on the network segment and ρ the fraction belonging to the requested service category. If devices reply only when they provide the requested service, then $|R| = \rho P$. If every device replies to every probe, then $|R| = P$. The same model therefore applies to both architectures, with the response strategy determining the value of $|R|$.

Across all evaluated scenarios, the measured traffic followed the linear model of Equation 7.1. In the discovery phase, the proposed mechanism received 235, 470, 942, 1882 and 3762 B for one, two, four, eight and sixteen responses, while the baseline received 2544, 5088, 10178, 20352 and 40700 B over the same range (see the cumulative difference at each iteration, including metadata download in Table 7.5). During metadata retrieval, each TD transferred over CoAP required 523 B, whereas each WSDL document transferred over HTTP required 2314 B. The resulting models are

$$C_{\text{proposed}} = 235\,|R| + 523\,k \qquad \text{and} \qquad C_{\text{baseline}} = 2544\,|R| + 2314\,k \tag{7.2}$$

where C is the received traffic in bytes. Over the measured range, the fitted model differed from the observations by at most two bytes. These coefficients, however, are specific to the evaluated implementation, protocol stack, document size and hardware, and should not be interpreted as architecture-independent constants.

Replacing $|R|$ with the expressions derived from Equation 7.1 gives

$$C_{\text{proposed}} = 235\,\rho P + 523\,k \qquad \text{and} \qquad C_{\text{baseline}} = 2544\,\rho P + 2314\,k \tag{7.3}$$

making explicit that both architectures scale with the fraction of relevant devices, because both filter the probe at the responder. What separates them is no longer where the first cut happens but what each response costs. The ratio between the two is therefore bounded, from 6.4 for a complete discovery at the measured point to 10.8 per added device of the requested category, and it reflects serialisation and transport rather than the number of devices that answer.

The coefficients 2544 B and 235 B depend partly on the length of the instance names and service types carried in each response. They reflect the identifiers chosen for this testbed, not a property of either protocol, and a deployment with longer identifiers would change both.

Assuming a single probe response ($|R| = 1$), the same model estimates how many metadata documents can be retrieved before the accumulated traffic matches that of one baseline discovery:

$$k^{\star} = \frac{(2544 + 2314) - 235}{523} \approx 8.8 \tag{7.4}$$

indicating that approximately eight complete TD can be retrieved within the traffic budget of a single baseline exchange. In the experiments, only one document was required in every discovery, so this limit was not reached. It is reported as an indication of the available margin for semantic evaluation rather than an observed operating point.

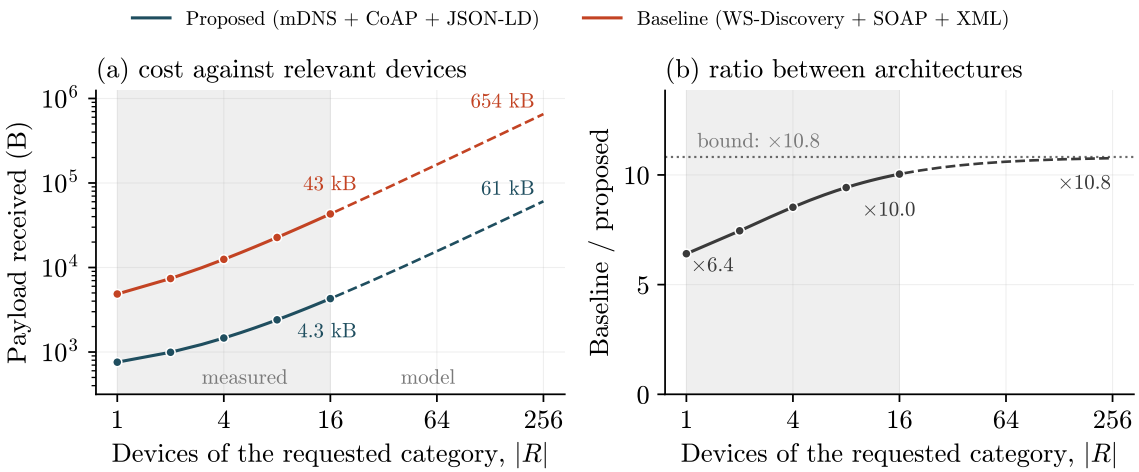


Figure 7.7: The model of Equation 7.3 evaluated against the number of devices of the requested category. The shaded region is the range the campaign covered and the dashed continuation is the model

Figure 7.7 compares both models against $|R| = \rho P$. A device outside the requested category responds to neither mechanism, so the population size drops out of both costs and the comparison depends only on the number of relevant devices. With one relevant device, the proposed approach generated 6.4 times less traffic than the baseline. The ratio grows with the number of relevant devices, reaching 10.0 at sixteen, and cannot exceed 10.8, the ratio of the two per-response coefficients. Beyond this point, however, the curves show how each approach scales and are not predictions of its performance.

The extrapolation assumes that the cost per response remains constant as the network grows, although this is unlikely in practice. As shown in Table 7.2, the baseline already exhibited unstable response behaviour with sixteen responders, and additional retransmissions in larger or more congested networks would increase the traffic beyond that predicted by the linear model. The extrapolation also assumes that the per-response cost of 2544 B remains constant as the number of relevant devices grows, which was measured only up to sixteen.

The projection also assumes that the underlying protocol stack and operating system behave similarly at larger scales, which was not verified experimentally. Finally, the analysis keeps $\rho = 1/16$ fixed. As the fraction of devices matching the requested service increases, the difference between the two approaches decreases, since $\rho \to 1$ makes both models depend on nearly the same number of responders. The projection therefore highlights the different scaling behaviour of the discovery strategies, not a fixed performance advantage at larger network sizes.

7.5 Compliance

The two applications described in Section 6.4.2 were deployed on separate hosts within the network segment characterised in Section 7.1, with the provider running on the Raspberry Pi and the consumer on the requester machine. The request specifies the `actuator` category and requires

`model=model_v3` and `version=1` as hard conditions during the first phase. In the second phase, it additionally requires that the provider be certified for the alloy `AA2024` and offer a positional accuracy not worse than 0.4 mm. As in the rest of the evaluation campaign, the probe window was set to 2 s. The experiment was repeated 41 times, with the first run discarded as a warm-up, leaving 41 samples for analysis.

7.5.1 The Cycle Exercised

By the end of the campaign, all 40 measured cycles had successfully completed the resolution, binding, and invocation stages. In every run, the provider was discovered, its document was retrieved and accepted, the binding was established, the service was invoked, and a result was returned by the provider's FB network. No cycle failed, no binding was established with a provider that did not satisfy the request, and every returned payload reported the requested alloy and seam length. Table 7.6 breaks down the execution time of each stage.

Interval	Description	Median (p10 – p90)
t_{disc}	the search, with a 2 s probe window	2019.6 (2018.6 – 2021.5)
t_{floor}	the search, window closed on the last answer	98.3 (55.0 – 136.6)
t_{bind}	establishing the binding	1.18 (0.74 – 1.30)
t_{rpc}	the invocation, end to end	67.2 (65.9 – 69.3)
t_{app}	of which inside the provider's application	56.0 (55.5 – 56.6)
t_{net}	of which network and protocol	11.1 (9.8 – 12.6)
$\mathbf{t}_{\text{total}}$	**from service request to the final invocation**	**2088.0**
$\mathbf{t}_{\text{total, floor}}$	**the same with the window closed**	**165.1** (127.1 – 203.8)

Table 7.6: One complete cycle, in milliseconds, over the 40 samples that follow the discarded warm-up. The simulated duration of the remote welding job is fixed at 50 ms and is contained in t_{app}.

The results show that the compliance cycle is dominated by the discovery probe window. With the timeout set to 2 s, approximately 95.1% of the total cycle is spent waiting, even though the final response needed to determine the outcome has already been received. This confirms the behaviour previously observed for the discovery mechanism in isolation (Section 7.3.2). Once this waiting period is excluded, the remaining operations take only 68.4,ms, of which 50 ms are the simulated service execution fixed at deployment time and 1.18 ms the establishment of the binding. Neither the binding nor the invocation therefore has a material effect on the completion time, which is dominated by the discovery protocol.

The measurements also show that service binding introduces minimum overhead. Converting a discovery result into a callable FB endpoint takes only 1.18 ms, accounting for less than 1% of the cycle when compared with the protocol floor. Also, the FB network is not a major source of latency. The provider application spends approximately 56.0 ms processing each request, with 50 ms attributable to the simulated workload. This leaves 5.89 ms for carrying the request through

the IEC 61499 runtime and returning the result.[2] Adding the CoAP exchange around it (11.1 ms), the mechanism contributes roughly 17 ms to each remote invocation on the evaluated hardware, excluding the duration of the service itself.

Finally, the independently measured protocol floor agrees with the value inferred from the packet captures of Section 7.3. The measured t_{floor} ranged from 45.9 to 148.3 ms, with a median of 98.3 ms, and these values are consistent with the random response delay RFC 6762 requires of responders answering shared-record queries [9]. The agreement matters because the two instruments share no code: one reads the wire, the other reads a clock inside an IEC 61499 application on a different pair of hosts.

7.5.2 Candidate Evaluation

The campaign in Section 7.5.1 was conducted with a single provider on the network segment. As a result, it demonstrates the discovery cycle but does not evaluate provider discrimination. To examine that behavior, a second welding robot, `weldbot-11`, is added alongside the original provider. Both robots belong to the same category, are located in the same zone, run the same model and version, and are idle. Consequently, the TXT record contains no information that distinguishes one from the other for this request. Both respond to the probe, both pass the prefilter, and the compensatory ranking initially favors the wrong provider, assigning it a score of 0.9989 against 0.7744, due to its shorter setup time and better reported health.

The second phase correctly rejects this provider for two reasons that cannot be represented in the compact TXT record. First, it is not certified for AA2024, and second, its positional accuracy of 0.6 mm does not satisfy the required 0.4 mm threshold. The consumer therefore binds to `weldbot-07` and successfully invokes it. The negative case also behaves as expected. When `weldbot-07` is removed, the remaining provider still responds to the probe and passes the prefilter, but after its document is retrieved, it is rejected. The function block raises `ERR`, no binding is established, and no invocation takes place. Rather than selecting a provider that cannot satisfy the request, the application correctly performs no action.

This experiment turns the argument made in Section 4.4 into an empirical observation. A mechanism relying only on the first phase would have bound to the wrong robot, whereas a mechanism relying only on the second phase would have retrieved a document from every device on the network segment. As reported in Section 7.4, this baseline approach incurs a cost of 2 544 B per device. By contrast, the additional cost of making the correct selection is a single extra CoAP exchange, amounting to 523 B and one additional round trip.

7.5.3 Instantiating the Discovered Block at Runtime

The cycle reported above binds a proxy block that is already present in the consumer application. The variant described in Section 6.4.3 removes that block from the drawing and lets `SEM_DEPLOY`

[2] Computed for each sample as t_{app} minus the execution time reported by the provider in the result payload, which was consistently 50 ms.

create it from the `cpps:fbType` field of the retrieved TD, wire it to the blocks that consume its output, and bind it, all while the application is running. It was exercised on the same testbed, with the same request and the same provider, so that the only difference from the previous cycle is where the proxy block comes from.

The funnel was identical in every cycle, with one responder, one viable candidate and one document retrieved, and all 41 cycles resolved, created the block, bound it and invoked the service, each returning the requested alloy and seam length. Table 7.7 places the two variants side by side.

Interval	Description	Static Proxy	Proxy created in runtime
t_{eff}	the search, window closed on the last answer	98.3	90.7
t_{bind}	establishing the binding	1.18	0.81
t_{rpc}	the invocation, end to end	67.2	67.1
t_{app}	of which inside the provider's application	56.0	56.0
	of which traversing the FB network	5.89	5.94
t_{net}	of which network and protocol	11.1	11.1
$t_{\text{total, eff}}$	the full cycle, window closed	165.1	160.2

Table 7.7: The cycle with the proxy block drawn in advance and with the proxy block created at runtime, in milliseconds, as medians over 41 samples in each campaign.

No statistically significant differences were observed between the two variants. A two sided test over the 41 samples of each campaign returned $p \geq 0.10$ for every metric, indicating that the measured differences are consistent with sampling variation. Even the largest difference, 7.7 ms in t_{eff}, falls within the approximately 90 ms interval introduced by the protocol's random delay.

This agreement is expected because both variants ultimately use the same binding mechanism. Rather than implementing a separate binding process, `SEM_DEPLOY` selects the appropriate proxy type from the `cpps:fbType` field, creates it through the IEC 61499 management commands when necessary, and otherwise falls back to `SEM_BIND`. The comparison therefore evaluates the same SIFB instantiated either at engineering time or during runtime, showing that runtime deployment produces behaviour equivalent to a statically configured application.

The deployment overhead itself is not reported because the `CREATE` and `WRITE` management commands were not included in the recorded traces. Even so, the deployment procedure creates a proxy only once and subsequent discoveries reuse the existing instance and update only its endpoint. As a result, rebinding after a provider change is less costly than the initial deployment.

Chapter 8

Discussion

Chapter 7 reports what was measured. This chapter reads it. It begins with the one behaviour that has to be understood before any of the timing figures can be interpreted, since it belongs to the protocol rather than to the architecture and is easily mistaken for the latter. It then uses the results to answer the two research questions stated in Section 1.3, and closes by setting out what the work does not establish, separating the limitations of the method from those of the implementation and from those of the design itself.

8.1 Intrinsic Behavior of the Protocols

The first phase of discovery is not slow because of the underlying mechanism. It takes time because both protocols intentionally delay their responses, although they do so in different ways. Without making that distinction, the latency figures in Section 7.3.2 could incorrectly suggest that the overhead comes from the architecture rather than from the behaviour the standards require. With a single provider on a wired segment, responses to probes arrived between 34 and 124 ms after the query, with a median of 84 ms, and varied noticeably across repeated measurements.

Since the network latency of the segment is well below one millisecond, as shown in Section 7.1, this interval cannot be explained by transmission delay. Instead, it reflects the random response delay that RFC 6762 requires mDNS responders to introduce when answering shared-record queries [9]. Independent packet captures measured delays ranging from 27 to 179 ms, while the compliance experiment, which measures the same quantity from within an application running on a different pair of hosts, recorded values between 45.9 and 148.3 ms. Although these three measurements use independent implementations and instrumentation, they converge on the same quantity.

The upper latency values require some qualification, since the measured maximum was 179 ms, exceeding the 120 ms delay defined by the protocol. This interval includes contributions that cannot be isolated by the measurement setup. The first is the responder delay, uniformly distributed between 20 and 120 ms. The second is the querier delay recommended by RFC 6762, which applies the same 20–120 ms random interval before the first query in a discovery sequence

to reduce synchronized transmissions. Because the measurements start at the application layer, this delay is included in the observed latency. The third contribution comes from scheduling effects on the responder, whose processor is limited to 600 MHz and therefore introduces additional timing variability.

The measured maximum is therefore consistent with the combined effect of these three factors and does not indicate non-compliance with the protocol. Instead, it shows that while the specification bounds the minimum random delay, the upper tail of the latency distribution also depends on the characteristics of the underlying hardware.

The consequence for the requester is a hard lower bound. A probe window shorter than 200 ms risks missing responders that behave exactly as the standard requires, and no compliant implementation can reduce this bound without violating the protocol. In this sense, the minimum latency of service discovery is dictated by the protocol rather than by implementation or system design. By contrast, the fixed probe window of 2 seconds adopted throughout the measurement campaign is purely a configuration choice. Under this setting, 95.1% of the discovery cycle is spent waiting after the last useful response has already been received, representing avoidable overhead for deployments in which the timeout can be safely reduced without compromising discovery process.

How the delay is introduced is what distinguishes the two systems as the number of responders increases. In mDNS, the requester draws a single random delay before sending the first query. The measured dispersion of 28.9 ms is consistent with what would be expected from a uniform draw over a 100 ms window, so adding responders has minor effect on the arrival time of the final response. On the other hand, with WS-Discovery, each responder draws its own delay, bounded at 500 ms and final response is therefore determined by the maximum of N independent random draws, causing the last arrival to increase by approximately 28 ms per responder.

The packet captures also revealed a behaviour that is not part of the design of the proposed approach but is intrinsic to the protocol it uses. Successive queries within the same probe window included previously received answers in their *known-answers* section, allowing providers whose records had already been advertised to remain silent. As a result, repeated mDNS queries do not introduce repeated cost, which helps explain why the gross over effective ratio in Table 7.4 is lower for the proposal than for the baseline. In addition, resolving each candidate required only 0.22 ms on average, because the SRV, TXT, and A records were delivered in the additional section of the original response rather than requiring follow-up queries.

8.2 Answering the Research Questions

8.2.1 Research Question 1

> *"How can dynamic and semantic resource discovery be achieved at runtime in a distributed CPPS while meeting the limitations of constrained devices and at what cost?"*

The question has two parts, and the results address each of them in turn.

How. The key idea is to move the initial semantic filtering from the consumer to the network itself. The capability category is encoded in the mDNS service type, while a compact capability summary is stored in the TXT record. As a result, devices that cannot satisfy a request never respond, and the full semantic description is retrieved only for the candidates that pass this initial filter. This is not simply a design claim, it is confirmed experimentally in Section 7.4. Adding fifteen devices from unrelated categories increased the traffic received by the requester by exactly zero bytes, across every composition tested and all forty repetitions of each experiment.

The second phase demonstrates that this filtering is genuinely semantic rather than merely syntactic. As Section 7.5.2 shows, both phases are necessary. Two robots that were indistinguishable based on every field available in the TXT record differed only in their retrieved descriptions. One advertised a certified-materials list and a typed physical quantity, while the other did not. The compensatory ranking therefore selected the incorrect robot. A discovery mechanism based solely on announcements would have bound to a robot incapable of welding the required alloy, whereas a mechanism relying only on semantic documents would have incurred the baseline traffic cost before reaching the same conclusion.

At what cost. The cost of the approach can be summarised by four measurements, all of which reflect properties of the architecture rather than of the experimental platform.

- **Traffic.** A complete discovery of a single provider transfers 828 B of payload, or 996 B on the wire, with identical results across every repetition. Each additional device in the requested category adds 235 B, while devices from other categories cost nothing at all. In a conformant baseline the corresponding costs are 2544 B and 0 B, respectively. Expressed as a function of the segment, the proposal costs 235 ρP + 523 k against 2544 ρP + 2314 k for the baseline: both scale with the relevant fraction of the segment, and the proposal is between 6.4 and 10.8 times cheaper depending on how many devices of the requested category answer.

- **Computation.** The requester spends $6.5 + 0.23\,|R|$ ms in the first phase, compared with $24.1 + 1.90\,|R|$ ms for the baseline. Evaluating a single semantic document requires 2.5 µs, compared with 22.2 µs in the baseline. Across every measured configuration, computation accounts for less than 11% of the total discovery time. Discovery is therefore dominated by waiting for the network rather than by computation, making the mechanism practical even for resource-constrained devices.

- **Latency.** The lower bound remains the protocol itself, at approximately 200 ms, as discussed in Section 8.1. The upper bound depends on the probe window configured for a particular deployment. The proposed architecture does not change this fundamental protocol limit.

- **Semantic headroom.** Equation 7.4 quantifies the available margin. The requester can retrieve and evaluate eight complete TDs before the accumulated traffic equals that of a single

baseline discovery involving one provider. In other words, the additional semantic expressiveness is achieved without consuming the bandwidth savings gained during the first phase.

The answer to Research Question 1 is therefore that semantic expressiveness and low-cost discovery on a distributed segment are not inherently competing objectives. Once discovery is divided into an initial filtering stage performed by the network and a semantic evaluation performed only on the remaining candidates, both can be achieved simultaneously. The measured cost of this design is less than one kilobyte of traffic and only a few milliseconds of processing per discovery, while the latency floor remains a property of the underlying protocol rather than of the architecture itself.

8.2.2 Research Question 2

"How may semantic discovery results be used to automate resource binding within an industry standard environment, such as an IEC 61499-compliant runtime?"

Section 6.4 answers this one by construction and Section 7.5 empirically. Three claims can be made from it.

The result of a discovery is consumed natively. The discovery process yields an endpoint together with an interface descriptor, and the standard already includes a construct specifically intended to transform this information into a running connection which is the SIFB. As a result, no translation layer, adapter, or runtime modifications were needed. Establishing the binding required only 1.18 ms, representing just 0.7% of a cycle, even when that cycle is measured with the probe window closed on the last relevant response. While the cost of discovery itself may be open to discussion, the process of consuming its result introduces essentially no additional overhead.

The service that is discovered is the service the runtime executes. The invocation does not end in a handler that merely happens to support CoAP. Instead, it enters the provider's application as an event, propagates through its function block network, and produces its result through a function block that is completely unaware of the discovery process. On a processor limited to 600 MHz, this traversal takes 5.89 ms, while the surrounding CoAP exchange adds 11.1 ms, resulting in approximately 17 ms of communication and runtime overhead, excluding the execution time of the service itself. This distinction is important since it demonstrates not only that the protocol functions correctly, but that a binding has been successfully established between two IEC 61499 applications.

The binding can be done at runtime. In the first variant, the proxy block is already part of the application, and only its endpoint is resolved at runtime, eliminating the need to define the device mapping during engineering. The second variant goes a step further by removing the proxy block entirely. Instead, the FB type is obtained from the retrieved TD document, instantiated in the running configuration using the standard's management commands, connected to the function blocks that consume its output, and bound without interrupting the application. Both approaches ultimately produce the same application behaviour. The key difference is that, in the second case, the application never needed to know at design time which type of function block it would invoke.

The overall result is clear, meaning that progressing from identifying a required capability to successfully invoking it took 2 088 ms with a 2 s probe window, and 165 ms when the window was closed on the last response that influenced the outcome. Of this total, the binding and invocation account for 68 ms, while the simulated job contributes 50 ms. The dominant source of latency is therefore the protocol itself rather than the architectural mechanisms introduced by the proposed approach. Consequently, the gap identified in Chapter 1, where an architecture that provides all the necessary conditions for distributed composition but lacks a mechanism to activate them at runtime, is addressed using only the constructs already defined by the standard.

8.3 What the Work Does Not Establish

The results above are bounded in three different ways, and the boundaries are worth separating because they carry different consequences. Some are properties of how the quantities were obtained, and would change if the campaign were run again differently. Some are properties of the prototype, and would change if it were rewritten. Some are properties of the architecture, and would require the architecture to change.

8.3.1 Limitations of the Method

The probe timeout dominates any measurement of elapsed time. Because a requester cannot know how many providers will respond, it must wait for a fixed timeout before proceeding, and that interval is always incurred in full. As a result, reporting the total discovery time primarily reflects the configured timeout rather than the performance of the discovery mechanism itself. For this reason, the evaluation separates the computation performed by the mechanism from the waiting period and reports the arrival times of responses instead. Although this provides a more meaningful representation of the mechanism's behaviour, it does not eliminate the underlying limitation. Comparisons of end-to-end discovery times with other work still depend on timeout values chosen independently for each implementation, making direct wall-clock comparisons generally unreliable.

The local mDNS cache had to be defeated. A probe issued shortly after a previous one may be satisfied from the local mDNS cache, making the discovery appear almost instantaneous and measuring the cache rather than the network. To avoid this effect, every discovery measurement was performed using a fresh resolver with a cold cache. Although this approach provides a more accurate characterisation of the discovery mechanism, it also represents a conservative scenario. In practice, deployments that perform repeated discoveries are likely to benefit from cached responses and achieve lower discovery times than those reported here. Consequently, the results presented in Chapter 7 should not be interpreted as the steady-state cost of a running system.

The byte counter does not capture kernel-generated traffic. The instrumentation counts only the packets generated by the application through the wrapped `send` and `receive` system calls, while accounting for protocol headers analytically: 14 B for Ethernet, 20 B for IPv4, and 8 B for UDP, or 20 B for TCP instead of the UDP header. As a result, it does not include the control

traffic generated by the TCP stack itself, such as the `SYN`, `SYN-ACK`, and `ACK` packets used during connection establishment, acknowledgements exchanged during data transfer, or the `FIN/ACK` sequence used to close the connection. This omission corresponds to approximately five to seven TCP segments of 54 B each, or between 270 and 378 B per HTTP transaction. Consequently, the reported byte count underestimates the communication cost of the HTTP baseline. This bias is conservative with respect to the proposed architecture, as it favours the baseline rather than inflating its overhead. Nevertheless, it does not affect the conclusions, since the measured communication cost of the baseline is already 6.4 times higher than that of the proposed approach at the measured point, and counting the omitted segments would raise that figure to roughly 6.8. The reported value should therefore be interpreted as a lower bound rather than an exact measurement

Two metrics were intentionally omitted. The selectivity of each discovery phase depends on the composition of the evaluated network rather than on the discovery mechanism. Reporting it would reflect the chosen experimental setup instead of the effectiveness of the proposed approach. Raw discovery time was also omitted for the reasons discussed previously.

Behaviour under network interference was not evaluated. Preliminary experiments with injected packet loss and competing multicast traffic were discarded. Consequently, this dissertation does not make claims about the behaviour of the proposed architecture under packet loss or heavy multicast contention, which remains an important limitation of the experimental campaign.

8.3.2 Limitations of the Implementation

The configuration files are not validated. A misspelled key produces no error and no output, only a device that comes up quietly incorrect. A schema would catch this at load time and remains a piece of work that was not done.

The CoAP transport is platform dependent. The `aiocoap` library selects its transport implementation according to the capabilities of the underlying platform. In particular, the transport that binds a single socket to a dual-stack any-address is not available on all systems. When this transport is unavailable, the fallback implementation does not support binding to an any-address. To ensure portability, the prototype detects the available transport at runtime and selects a configuration supported by the host platform.

The cache described in Section 4.5 was not built. A cache that expires only by a TTL is not faithful to this problem, because a device can leave the network at any moment and a purely time-based entry would keep pointing at it until it aged out. Doing it properly requires a browser that outlives an individual search together with a mechanism to detect departures, which is a separate piece of design. Since the fixed probe window dominates every cycle measured, this is also the single change with the largest effect on the numbers reported here, and its absence makes those numbers pessimistic rather than optimistic.

8.3.3 Limitations of the Design

The vocabulary is assumed to be shared. The hierarchical service type encoded in the mDNS name, the identifiers stored in the TXT record and the semantic extension used by TD documents rely on consumers and providers sharing a common vocabulary for capability types. In this work, that vocabulary is assumed to be defined at deployment time for a given industrial domain. The maintenance of an ontology that evolves as new device types and capabilities are introduced into a plant remains a recognised open problem [29]. The proposed approach partially mitigates this issue because each device only needs to maintain the subset of the vocabulary relevant to its own functional profile, reducing both memory requirements and the complexity of updates. However, this does not address the more fundamental challenge of defining and governing the shared semantics of the vocabulary itself.

The probe timeout is unavoidable. Since a requester does not know how many providers will respond, it must wait for the full timeout before completing discovery. This makes the timeout a trade-off between responsiveness and the risk of missing valid responses. As discussed in Section 8.1, the protocol defines a lower bound for this interval, but not an optimal value. This limitation is inherent to one-shot multicast discovery and motivates the use of caching.

The evaluation is limited to a single network segment. In its standard configuration, mDNS operates only within a local link. Deployments spanning multiple Virtual Local Area Networks (VLANs) or subnets require additional mechanisms, such as mDNS proxies or repeaters. Since all experiments were conducted on a single segment, no claims are made about behaviour across subnet boundaries.

The announcement layer is unauthenticated. mDNS provides no authentication of announcements, so a malicious device can impersonate a legitimate provider, and a consumer that filters on a TXT record filters on a claim. Securing the CoAP side is well understood and supported in constrained environments through Datagram Transport Layer Security (DTLS).

TXT records carry text and nothing else. A configuration file may write `version: 1` and mean the number, but what arrives from the network is the string. Hard conditions over the TXT record are therefore textual comparisons, which is an accurate acknowledgement of what the transport is, and also means that no ordering or arithmetic relation can be expressed in the first phase. Anything numeric that has to be compared as a number belongs to the second.

Chapter 9

Conclusion

This dissertation addresses a standing limitation in distributed control systems. Modern architectures, particularly IEC 61499, separate application logic from the hardware that executes it, creating the conditions for applications to be composed and reconfigured at runtime. However, they do not provide a standardised mechanism for discovering, during operation, which device on the network can perform a given task. As a result, the mapping between application logic and physical devices is typically defined during engineering and remains fixed after deployment.

Chapter 3 showed that this limitation is inherent to the discovery process. Descriptions detailed enough to capture device capabilities are too large to be included in discovery announcements, leaving two inefficient alternatives: retrieving descriptions from every device or relying on a central repository. The approach proposed in Chapter 4 addresses this by separating discovery into two stages. A compact summary is included in the announcement, allowing most devices to be filtered before any metadata is retrieved, while the full semantic description is requested only from the remaining candidates. As a result, devices that do not match the requested service type do not respond to the discovery query. Chapter 5 implemented this approach using standard mDNS and CoAP without modifying either protocol, and Chapters 6 and 7 evaluated it against the discovery mechanism commonly used in industrial SOA under identical experimental conditions.

9.1 Contributions

The original contributions of this work, as anticipated in Section 1.4, are the following.

1. **A two-phase distributed discovery architecture** that performs both syntactic and semantic matching with no registry infrastructure, by encoding the service category in the mDNS service type, carrying a compact semantic summary in the TXT record, and retrieving a full semantic description over CoAP only for the candidates the first phase did not remove. The specific claim, and the one the results support, is that the coarse cut belongs in the network rather than in the consumer.

2. **A prototype implementation** that realises the architecture on unmodified standard protocols and records the design decisions that only became apparent once there was code to run, among them the textual nature of TXT records, the set-relative normalisation the ranking requires, and the platform dependence of the CoAP transport.

3. **A measurement testbed and instrumentation** built so that every reported quantity is a property of the architecture rather than of the platform. The design of the testbed follows from three empirical constraints, chief among them that co-located responders collapse the arrival distribution and cannot be used to characterise a distributed mechanism, which is a result about how such mechanisms must be measured and not only about this one.

4. **A quantitative comparison against WS-Discovery with SOAP-over-HTTP**, conducted on the same testbed with the same population and the same request, together with the scaling coefficients that allow the comparison to be extended beyond the measured points. The comparison yields two expressions, $235\,\rho P + 523\,k$ against $2544\,\rho P + 2314\,k$ bytes received over the measured range, in which the fraction of the segment that could answer the request appears in one and not in the other.

5. **A realisation of the discovery result as an IEC 61499 binding**, in which the endpoint and interface descriptor a discovery produces are consumed by SIFBs in an unmodified compliant runtime, the invocation is executed by the provider's own function block network, and the proxy block itself can be created and wired into a running configuration from the retrieved description.

9.2 Achievement of the Objectives

The four objectives set out in Section 1.3 can now be checked against evidence rather than against intent. Table 9.1 states each objective, the measurement that bears on it, and where that measurement is reported.

Objective	Evidence	Reported in
Fully distributed	No component holds knowledge of the population and no participant is configured with the address of another; the only shared state is the multicast group both protocols already use. A provider joins by announcing and leaves by withdrawing, and both were exercised.	Sec. 7.4
Within reach of constrained devices	A complete discovery moves under a kilobyte and costs nothing for devices outside the requested category; the requester spends single-digit milliseconds per discovery and under 11% of elapsed time computing, and the whole provider population ran concurrently on a board capped at 600 MHz.	Sec. 7.3, 7.4
Matching on what a service can do	Two devices identical in every field an announcement can carry were separated only by the retrieved document, and the ranking over the announcement alone preferred the one that could not do the job, which shows that both phases are needed rather than either.	Sec. 7.5.2
Standard protocols, result consumed natively	No extension to mDNS, DNS-SD or CoAP was made or required, and a discovery result was turned into a running binding inside an unmodified IEC 61499 runtime, using the construct the standard already provides, in 1.18 ms.	Sec. 7.5

Table 9.1: The four objectives against the evidence produced by this dissertation.

The results should not be interpreted as universally applicable to all deployments. As discussed in Chapter 8, their extent depends on the characteristics of the evaluated system. In particular, the main benefit of the proposed approach increases with the heterogeneity of the network segment rather than representing a fixed performance gain.

9.3 The Target Environments

The results are most relevant to a specific class of systems rather than to distributed automation in general. The benefits of the proposed approach depend on the fraction of devices that cannot satisfy a given request, making it most effective in heterogeneous network segments.

This makes the approach particularly suitable for reconfigurable production cells with diverse device populations. It is also well suited to well consolidated deployments, as it does not require dedicated servers, protocol extensions or additional network infrastructure. The same characteristics make it attractive for mobile resources, such as tools shared between production cells, and for failover scenarios, where an application can rediscover and bind to an equivalent provider after a failure. The evaluation also shows that the approach is compatible with constrained edge devices,

requiring little processing or communication overhead while avoiding unnecessary responses from devices that do not provide the requested service.

The same observations also define its limitations. The mechanism is not intended for hard real-time control loops, since the discovery latency imposed by the underlying protocols makes it suitable for configuration and rebinding rather than cyclic control. It is also less appropriate for very large flat networks, where multicast scalability becomes a limiting factor. Finally, in networks where most devices provide the requested service, the advantage decreases because nearly all devices participate in the discovery process.

9.4 Scientific Impact

Three elements of this work are, in the author's judgement, transferable beyond the specific mechanism proposed.

The first is the architecture itself. The observation that semantic expressiveness and low distributed cost are opposed rests on the assumption that a consumer must obtain descriptions in order to discriminate. Splitting the decision into a cut the network performs and an evaluation the consumer performs breaks that assumption, and nothing about it is specific to mDNS, to CoAP or to welding robots. Any discovery protocol in which a responder decides whether to answer based on the content of the question admits the same construction. Table 3.1 placed this work in a cell that the reviewed literature leaves empty and the results give that placement a quantitative basis.

The second contribution is the measurement methodology. The evaluation was designed so that every reported metric reflects the behaviour of the discovery mechanism rather than characteristics of the experimental platform or setup. This principle also motivated the exclusion of metrics such as selectivity, which depends on the composition of the evaluated network, and raw discovery time, which is dominated by protocol timeouts. In addition, the observation that co-located responders alter the response arrival distribution is a methodological contribution in itself, showing that this common testbed configuration can produce misleading performance results.

The third contribution is the comparison model. Representing the communication cost of both architectures as $\beta_1|R| + \beta_2 k$, with |R| equal to ρP for both when each filters the probe at the responder, separates what a discovery costs per answering device from how many devices answer, and shifts the comparison from implementation-specific measurements to the two coefficients that distinguish the stacks. While the numerical coefficients are specific to the evaluated platform, protocol libraries, document sizes and experimental conditions, the dependency captured by the model is independent of those factors. The coefficients are therefore intended as a description of the measured system and as a basis for future comparisons.

9.5 Future Work

From link-local to enterprise scope. A natural extension of this work is to expand the proposed discovery mechanism beyond a single network segment. Since mDNS and unicast DNS-SD share

the same service discovery model, the approach can be integrated with standard DNS infrastructure without modifying its design. This would enable semantic discovery across VLANs, production cells and enterprise sites while preserving the same service types and TXT records. It would also allow cloud-based services, such as analytics or optimisation applications, to be discovered using the same mechanism as local devices. Extending the approach in this direction raises new challenges related to system administration and security, including automated service registration, zone management and the trust model required when discovery is no longer restricted to a local network.

Richer first-phase filtering through hashed semantic identifiers. The first phase is currently limited by what fits in a TXT record and by the fact that it relies on string matching. Encoding semantic identifiers as a compact probabilistic structure, such as a Bloom filter, over the terms describing a capability would allow a single small field to represent an arbitrary number of semantic terms, while enabling a consumer to test whether the required terms are present before retrieving any metadata document. This generalises the Bloom filter-based representation proposed by Amoretti et al. [2] from matching a node or service description to matching a set of semantic terms describing a capability. It would also make the first phase both more selective and more informative, since the number of matching terms could be used to rank candidates rather than simply accepting or rejecting them. The trade-off is well understood and quantifiable: each false positive incurs the cost of one additional second-phase retrieval, and Equation 7.4 already provides the available communication budget for selecting the Bloom filter parameters.

The cache, done properly. A cache that expires only by TTL is not faithful to a network where devices can leave at any moment. Building it correctly means a browser that outlives an individual search and a mechanism for detecting departures, for which CoAP Observe on the provider's metadata endpoint is the natural instrument [23]: an abrupt departure ends the notification stream and invalidates the entry. Since the fixed probe window is 95% of every cycle measured in this work, this is also the single change with the largest effect on the numbers reported here.

Authenticated announcements. Securing the CoAP side is well understood through DTLS. Securing the announcement layer is the harder half, and until it is addressed a consumer filtering on a TXT record is filtering on an unverified claim. Any credible industrial deployment of this mechanism needs an answer here.

Completing the interference campaign. The behaviour of the mechanism under link loss and under a busy multicast group was not measured, for the reasons given in Section 8.3.1, and it is the most conspicuous gap in the validation. A retry policy that respects the protocol floor rather than fighting it is the obvious companion piece of work.

Deeper matchmaking within a known budget. The semantic evaluation used here is a direct comparison of typed values against requirements. Subsumption reasoning over a shared ontology would be more expressive, and the argument against it in constrained environments has always been cost. That argument can now be made quantitatively rather than rhetorically: evaluation of one document costs 2.5 µs against a discovery that spends most of its time waiting, and the traffic budget measured here allows roughly eight documents to be examined before the mechanism costs

what one baseline exchange costs. There is room to spend, and this work gives a first estimate of how much.

References

[1] Meriem Achir et al. “Service discovery and selection in IoT: A survey and a taxonomy”. In: *Journal of Network and Computer Applications* 200 (Apr. 2022).

[2] Michele Amoretti et al. “DINAS: A Lightweight and Efficient Distributed Naming Service for All-IP Wireless Sensor Networks”. In: *IEEE Internet of Things Journal* 4 (3 June 2017), pp. 670–684.

[3] Christian Amsüss et al. *Constrained RESTful Environments (CoRE) Resource Directory*. RFC 9176. Apr. 2022.

[4] Apple Inc. *Bonjour Overview*. Available at `https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/NetServices/Introduction.html`. Apple Developer. 2013.

[5] Carsten Bormann and Zach Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. Aug. 2016.

[6] Arne Broering, Soumya Kanti Datta, and Christian Bonnet. “A Categorization of Discovery Technologies for the Internet of Things”. In: *6th International Conference on the Internet of Things*. 2016, pp. 131–139.

[7] Gonçalo Cândido et al. “Service-oriented infrastructure to support the deployment of evolvable production systems”. In: *IEEE Transactions on Industrial Informatics* 7 (4 2011), pp. 759–767.

[8] Stuart Cheshire and Marc Krochmal. *DNS-Based Service Discovery*. RFC 6763. Feb. 2013.

[9] Stuart Cheshire and Marc Krochmal. *Multicast DNS*. RFC 6762. Feb. 2013.

[10] Simone Cirani et al. “A scalable and self-configuring architecture for service discovery in the internet of things”. In: *IEEE Internet of Things Journal* 1 (5 Oct. 2014), pp. 508–521.

[11] Armando W Colombo et al. *Industrial Cloud-Based Cyber-Physical Systems: The IMC-AESOP Approach*. 1st ed. Springer Cham, 2014.

[12] Wenbin Dai, Wanqi Huang, and Valeriy Vyatkin. “Enabling plug-and-play software components in industrial cyber-physical systems by adopting service-oriented architecture paradigm”. In: *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*. IEEE Press, 2016, pp. 5253–5258.

[13] Wenbin Dai et al. “Bridging service-oriented architecture and IEC 61499 for flexibility and interoperability”. In: *IEEE Transactions on Industrial Informatics* 11 (3 June 2015), pp. 771–781.

[14] Wenbin Dai et al. “Design of Industrial Edge Applications Based on IEC 61499 Microservices and Containers”. In: *IEEE Transactions on Industrial Informatics* 19 (7 July 2023), pp. 7925–7935.

[15] Wenbin Dai et al. “Semantic Integration of Plug-and-Play Software Components for Industrial Edges Based on Microservices”. In: *IEEE Access* 7 (2019), pp. 125882–125892.

[16] Evgenii Demin et al. “IEC 61499 Distributed Control Enhanced with Cloud-based Web-Services”. In: *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2015, pp. 972–977.

[17] Simon Fernandez et al. “Semantic Identifiers and DNS Names for IoT”. In: *2021 International Conference on Computer Communications and Networks (ICCCN)*. July 2021, pp. 1–9.

[18] Roy Thomas Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. University of California, 2000.

[19] Eclipse Foundation. *4diac FORTE – IEC 61499 Runtime Environment*. 2026.

[20] Eclipse Foundation. *Eclipse 4diac – An Open Source Environment for Distributed Industrial Automation and Control Systems*. 2026.

[21] Google Inc. *Google Cast Overview*. Available at https://developers.google.com/cast/docs/overview. Google Developer. 2013.

[22] Michel Grabisch. “The application of fuzzy integrals in multicriteria decision making”. In: *European Journal of Operational Research* 89 (3 Mar. 1996), pp. 445–456.

[23] Klaus Hartke. *Observing Resources in the Constrained Application Protocol (CoAP)*. RFC 7641. Sept. 2015.

[24] IEC. *OPC Unified Architecture - Part 1: Overview and Concepts*. Geneva, Switzerland: International Electrotechnical Commission, 2020.

[25] IEC. *OPC Unified Architecture - Part 12: Discovery and Global Services*. Geneva, Switzerland: International Electrotechnical Commission, 2020.

[26] International Electrotechnical Commission. *IEC 61499-1:2012 Function Blocks - Part 1: Architecture*. International Standard, et seq. Geneva, Switzerland, 2012.

[27] Ahmed Ismail and Wolfgang Kastner. “Discovery in SOA-Governed Industrial Middleware with mDNS and DNS-SD”. In: *21st International Conference on Emerging Technologies and Factory Automation*. IEEE, 2016, pp. 1–8.

[28] François Jammes and Harm Smit. “SERVICE-ORIENTED PARADIGMS IN INDUSTRIAL AUTOMATION”. In: *IEEE Transactions on Industrial Informatics* 1 (Feb. 2005), pp. 62–70.

[29] Vaclav Jirkovsky et al. “Toward Plug&Play Cyber-Physical System Components”. In: *IEEE Transactions on Industrial Informatics* 14 (6 June 2018), pp. 2803–2811.

[30] Sebastian Käbisch, Michael McCool, and Ege Korkan. *Web of Things (WoT) Thing Description 1.1*. W3C Recommendation. Dec. 2023.

[31] Edward A Lee. *Cyber Physical Systems: Design Challenges*. Tech. rep. Jan. 2008.

[32] Chun Feng Liao and Yu Jou Weng. “Enabling Space-Aware Service Discovery Model in Home Networks through a Compatible Extension to mDNS/DNS-SD”. In: *Electronics (Switzerland)* 12 (18 Sept. 2023).

[33] Bengt Lindberg, Mauro Onori, and D Semere. “Evolvable production systems: A position paper”. In: *Swedish Production Symposium, Gothenburg, Sweden*. 2007.

[34] Andreas Lober, Hartwig Baumgärtel, and Richard Verbeet. “Semantic service discovery in heterogeneous cyber-physical systems”. In: *Hamburg International Conference on Logistics (HICL)*. Sept. 2020.

[35] Yan Lu and Feng Ju. “Smart Manufacturing Systems based on Cyber-physical Manufacturing Services (CPMS)”. In: *20th IFAC World Congress*. Vol. 50. Elsevier B.V., July 2017, pp. 15883–15889.

[36] L. Monostori et al. “Cyber-physical systems in manufacturing”. In: *CIRP Annals* 65 (2 2016), pp. 621–641.

[37] Quang Duy Nguyen et al. “Towards a Web-of-Things Approach for OPC UA Field Device Discovery in the Industrial IoT”. In: *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*. Vol. 2022-April. Institute of Electrical and Electronics Engineers Inc., 2022.

[38] Naghmeh Niknejad et al. *Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation*. July 2020.

[39] Hye-Young Paik et al. *Web Service Implementation and Composition Techniques*. 1st ed. Springer Cham, 2018.

[40] Mike P. Papazoglou and Willem Jan Van Den Heuvel. “Service oriented architectures: Approaches, technologies and research issues”. In: *VLDB Journal* 16 (3 July 2007), pp. 389–415.

[41] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. “Proceedings of the 17th international conference on World Wide Web”. In: *Proceeding of the 17th International Conference on World Wide Web 2008, WWW’08*. Apr. 2008, pp. 805–814.

[42] Eliseu Pereira, Joao Reis, and Gil Gonçalves. “DINASORE: A Dynamic Intelligent Reconfiguration Tool for Cyber-Physical Production Systems”. In: *Eclipse Conference on Security, Artificial Intelligence, and Modeling for the Next Generation Internet of Things (Eclipse SAM IoT)*. 2020, pp. 63–71.

[43] Stefan Profanter et al. “OPC UA for plug & produce: Automatic device discovery using LDS-ME”. In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017, pp. 1–8.

[44] Michele Ruta et al. “CoAP-based collaborative sensor networks in the Semantic Web of Things”. In: *Journal of Ambient Intelligence and Humanized Computing* 10 (Mar. 2018), pp. 2545–2562.

[45] B P Santos et al. “Indústria 4.0: Desafios e Oportunidades”. In: *Revista Produção e Desenvolvimento* (1 2018), pp. 111–124.

[46] National Academy of Science and Engineering of Germany. *Recommendations for implementing the strategic initiative INDUSTRIE 4.0. Final report of the Industrie 4.0 Working Group*. 2013.

[47] Zach Shelby. *Constrained RESTful Environments (CoRE) Link Format*. RFC 6690. Aug. 2012.

[48] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014.

[49] Peter Van der Stok, Michael Koster, and Christian Amsüss. *CoRE Resource Directory: DNS-SD mapping*. Internet-Draft draft-ietf-core-rd-dns-sd-05. Work in Progress. Internet Engineering Task Force, July 2019. 14 pp.

[50] Klaus-Dieter Thoben, Stefan Wiesner, and Thorsten Wuest. ""Industrie 4.0" and Smart Manufacturing – A Review of Research Issues and Application Examples". In: *International Journal of Automation Technology* 11 (1 2017), pp. 4–16.

[51] Duc Do Tran et al. "Towards Time-Sensitive Behavioral Contract Monitors for IEC 61499 Function Blocks". In: Sept. 2020.

[52] W3C. *Devices Profile for Web Services Version 1.1*. 2009.

[53] W3C. *Web Services Architecture*. 2004.

[54] W3C. *Web Services Dynamic Discovery (WS-Discovery) Version 1.1*. 2009.

[55] Ali Yachir et al. "Semantic resource discovery with COAP in the internet of things". In: *ICETE 2017 - Proceedings of the 14th International Joint Conference on e-Business and Telecommunications*. Vol. 6. SciTePress, 2017, pp. 75–82.

[56] Elmar Zeeb et al. "Service-Oriented Architectures for Embedded Systems Using Devices Profile for Web Services". In: *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*. 2007, pp. 956–963.

[57] Su Zhang. "Apply SOA Paradigms in Cyber-Physical System to Enhance Interoperability: State-of-the-Art Review". In: *arXiv preprint arXiv:1903.00065* (Mar. 2019).