A Vector Symbolic Approach for Cognitive Services and Decentralized Workflows

A thesis submitted in partial fulfilment
of the requirement for the degree of Doctor of Philosophy

Christopher Simpkin

December 2021

Cardiff University
School of Computer Science & Informatics

To my wife
Ann

For her infinite patience and everlasting support.
She has always sacrificed to enable my dreams.

Abstract

The proliferation of smart devices and sensors known as the Internet of Things (IoT), along with the transformation of mobile phones into powerful handheld computers as well as the continuing advancement in high-speed communication technologies, introduces new possibilities for collaborative distributed computing and collaborative workflows along with a new set of problems to be solved.

However, traditional service-based applications, in fixed networks, are typically constructed and managed centrally and assume stable service endpoints and adequate network connectivity. Constructing and maintaining such applications in dynamic heterogeneous wireless networked environments, where limited bandwidth and transient connectivity are commonplace, presents significant challenges and makes centralized application construction and management impossible.

The key objective for this thesis can be summarised as follows: a means is required to discover and orchestrate sequences of micro-services, i.e., workflows, on-demand, using currently available distributed resources (compute devices, functional services, data and sensors) in spite of a poor quality (fragmented, low bandwidth) network infrastructure and without central control. It is desirable to be able to compose such workflows *on-the-fly* in order to fulfil an *'intent'*.

The research undertaken investigates how service definition, service matching and decentralised service composition and orchestration can be achieved without centralised control using an approach based on a Binary Spatter Code Vector Symbolic Architecture and shows that the approach offers significant advantages in environments where communication networks are unreliable.

The outcomes demonstrate a new cognitive workflow model that uses one-to-many communications to enable intelligent cooperation between self-describing service entities that can self-organise to complete a workflow task. Workflow orchestration overhead was minimised using two innovations, a local arbitration mechanism that uses a delayed response mechanism to suppress responses that are not an ideal match and the holographic nature of VSA descriptions enables messages to be truncated without loss of meaning. A new hierarchical VSA encoding scheme was created that is scaleable to any number of vector embeddings including workflow steps. The encoding can also facilitate learning since it provides unique contexts for each step in a workflow. The encoding also enables service pre-provisioning because individual workflow steps can be decoded easily by any service receiving a multicast workflow vector.

This thesis brings the state-of-the-art closer to the ability to discover distributed services *on-the-fly* to fulfil an intent and without the need for centralised management or the imperative definition of all service steps, including locations. The use of a mathematically deterministic distributed vector representation in the form of BSC vectors for both service objects and workflows enables a common language for all elements required to discover and execute workflows in decentralised transient environments and opens up the possibilities of employing learning algorithms that can advance the state-of-the-art in distributed workflows towards a true cognitive distributed network architecture.

Acknowledgements

I would very much like to thank the following people for their guidance and support; Prof. Ian Taylor, my primary supervisor, Prof. Alun Preece, my secondary supervisor and Dr Graham Bent for his great insights and unending enthusiasm in this research area.

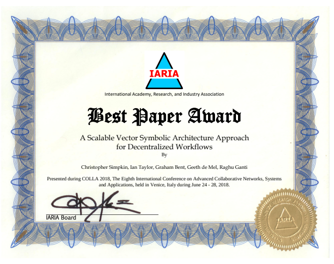
This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. Part of this work was also supported in part by the European Research Council through the European Union's Horizon 2020 Research and Innovation Program under Grant 682675 and in part by the European Union's Horizon 2020 Research and Innovation Program through the project MNEMOSENE under Grant 780215.

Contents

List of Publications

The work introduced in this thesis is based on the following publications.

- Chris Simpkin, Ian Taylor, Graham A Bent, Geeth De Mel, and Swati Rallapalli. Decentralized microservice workflows for coalition environments. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 2017
- Christopher Simpkin, Ian Taylor, Graham A Bent, Geeth de Mel, and Raghu K Ganti. A scalable vector symbolic architecture approach for decentralized workflows. In *COLLA 2018 The Eighth International Conference on Advanced Collaborative Networks, Systems and Applications*, pages 21–27. IARIA, 2018


IARIA

International Academy, Research, and Industry Association

Best Paper Award

A Scalable Vector Symbolic Architecture Approach
for Decentralized Workflows

By

Christopher Simpkin, Ian Taylor, Graham Bent, Geeth de Mel, Raghu Ganti

Presented during COLLA 2018, The Eighth International Conference on Advanced Collaborative Networks, Systems and Applications, held in Venice, Italy during June 24 - 28, 2018.

[illegible]

IARIA Board

- Christopher Simpkin, Ian Taylor, Daniel Harborne, Graham Bent, Alun Preece, and Ragu K Ganti. Dynamic distributed orchestration of node-red iot workflows using a vector symbolic architecture. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 52–63. IEEE, 2018

  – Creation and description of the binding scheme and methods for encoding service descriptions and encoding JSON service descriptions into VSA representations.

- Chris Simpkin, Ian Taylor, Graham A Bent, Geeth de Mel, Swati Rallapalli, Liang Ma, and Mudhakar Srivatsa. Constructing distributed time-critical applications using cognitive enabled services. *Future Generation Computer Systems*, 100:70–85, 2019

- Chris Simpkin, Ian Taylor, Daniel Harborne, Graham Bent, Alun Preece, and Raghu K Ganti. Efficient orchestration of node-red iot workflows using a vector symbolic architecture. *Future Generation Computer Systems*, 111:117–131, 2020

- Graham Bent, Christopher Simpkin, Ian Taylor, Abbas Rahimi, Geethan Karunaratne, Abu Sebastian, Declan Millar, Andreas Martens, and Kaushik Roy. Energy efficient 'in memory' computing to enable decentralised service workflow composition in support of multi-domain operations. In Tien Pham and Latasha Solomon, editors, *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III*, volume 11746, pages 414 – 435. International Society for Optics and Photonics, SPIE, 2021. doi: 10.1117/12.2586988. URL `https://doi.org/10.1117/12.2586988`

‘

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ACT-R	Adaptive Control of Thought-Rational
AI	Artificial Intelligence
ANN	Artificial Neural Network
BSC	Binary Spatter Code
CERN	The European Organisation for Nuclear Research
CCS	Cognitive Computing System
DAG	Directed Acyclic Graph
DAIS	The International Technology Alliance in Distributed Analytics and Information Sciences
DTFT	discrete-time Fourier transform
DHT	Distributed Hash Table
DNS	Domain Name Service
DNS-SD	DNS Service Discovery
DS	Discovery Service
EPC	Electronic Product Code

EPCIS	EPC Information Services
FMN	Federated Mission Networks
GPS	Global Positioning System
HD	Normalised Hamming Distance
HDS	Normalised Hamming Similarity
HPC	High-Performance Computing
HRR	Holographic Reduced Representation
HUMINT	Human intelligence
I.I.D	independent and identically distributed
IoT	Internet of Things
IoBT	Internet of Battlefield Things
IPP	Initial Program Plan
ISR	Intelligence, surveillance, and reconnaissance
LDAP	Lightweight Directory Access Protocol
MANET	mobile ad hoc network
MDO	Multi-Domain Operations
MGRS	Military Grid Reference System
MMO	Massively Multiplayer Online Game
MOD	U.K. Ministry of Defence
NLP	Natural Language Processing

OWL 2 Web Ontology Language 2

OWL-S Web Ontology Language for Services

P2P Peer-to-Peer

PCM phase-change memory

QoS Quality of Service

RFID Radio-frequency identification

RPC Remote Procedure Call

RR Resource Record

RESR Representational state transfer

SaaS Software as a Service

SEO Search Engine Optimisation

SOA Service Oriented Architecture

SPAUN Semantic Pointer Architecture Unified Network

SV Service Vector

SVA Service Vector Architecture

SWS Semantic Web Service

TacCIS Tactical Communications and Information Systems

ToB Tower of Babel problem

UUID Universally Unique Identifier

VSA Vector Symbolic Architecture

W3C	The World Wide Web Consortium
WA	Workflow-Agent
WFMS	Workflow Management System

Chapter 1

Introduction

This chapter sets the background context for the key problems that this thesis attempts to address, which feed into a hypothesis that clearly outlines the research goals. This chapter gives a brief history of the reasons why distributed computing has become the most powerful architectural approach to data analysis and transaction processing. I then outline some of the fundamental issues that must be overcome in distributed computing, relating these issues to existing and emerging distributed computing paradigms and supporting technologies, including workflows and Workflow Management System (WFMS). I then argue that the continuing advances in high-speed networking and proliferation of smart devices lead to a requirement for a new workflow methodology capable of discovering and executing workflows in dynamic, decentralized transient network environments; leading to my hypothesis which states how this can be achieved using a Vector Symbolic Architecture (VSA).

1.1 Why data analytics

Data analytics in its basic form has been practised from ancient times. For example, the Mayan civilisation attempted to predict harvests based on current and historical behaviour of the climate and, albeit an incorrect belief, the configuration of the constellations. The essential benefit gained from data analytics is the potential to correctly predict some aspect of the future based on records of historical events as well as data representing the current situation. As our ability to understand the world and its workings becomes more sophisticated (including human behaviour), we naturally start to consider more and more of this understanding to make better predictions. By the time of the Second World War and advent of computers, data analytics was used to analyse and decipher streams of opposition communications and gain a tactical advantage.

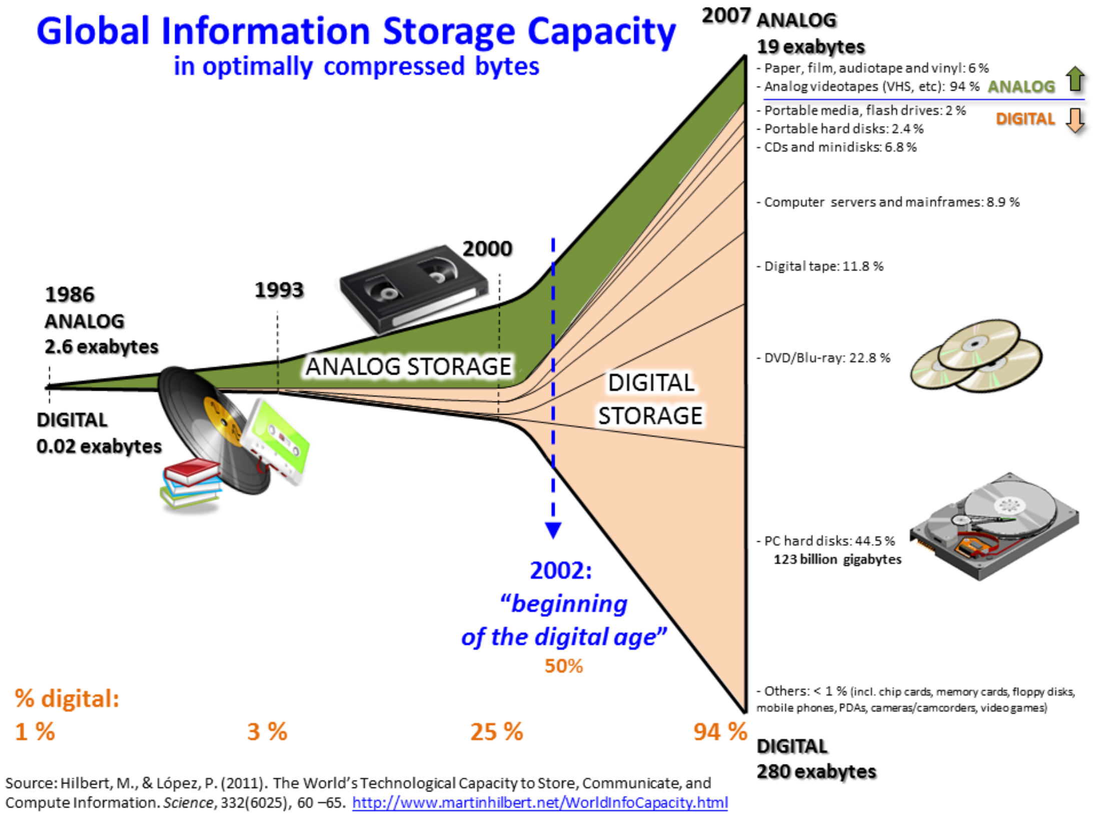


Figure 1.1: Exponential growth of worldwide data availability.

In modern times, data is available and continuously being generated on enormous scales. The European Organisation for Nuclear Research (CERN) currently generates around 2.5 petabytes of data daily[7]. Twitter handles an estimated 500 million tweets per day[8]; Google handles 3.5 billion internet search queries per day[9]. Figure 1.1 shows Hilbert & López's estimate of the world's capacity to store data, which, in 2007, was determined to be approximately 300 exabytes[10]. By 2014, due to the exponential nature of data generation, this estimate was revised by Hilbert to 4.7 zetabytes[11].

Machine learning techniques leverage the billions of data samples, available via the internet, to build statistical models that enable computers to outperform human operators at specialist pattern matching tasks such as cancer cell detection and have enabled computers to understand natural language commands, including words spoken by people in many accents and dialects.

Such massive data availability requires multiple computer devices and parallel processing in order to obtain useful analysis in a sensible time-frames. This has led to the advent of the fields of distributed computing and workflow formalisations.

1.2 Distributed Computing

The result of any data analytics task, for example, the possibility of rain tomorrow, must be arrived at in a timely fashion otherwise it becomes obsolete. When very large volumes of data must be processed, timely results can only be achieved by partitioning the data and processing each piece in parallel. Two mechanisms have emerged to facilitate this. Modern supercomputers[12] employ many thousands of tightly coupled processors together with shared memory and storage to achieve the necessary data processing speeds required for analysis of computationally intensive tasks such as weather forecasting, quantum mechanics and molecular modeling. This was the mainstay of all data and compute-intensive tasks until the advent of high-speed communication technologies, that enabled multiple, physically separate, computers to participate in data analytics tasks by message passing. Combined with the relentless fall in computer hardware prices, this has led to an alternate approach, namely, Distributed Computing.

An essential difference between supercomputing and distributed computing is that supercomputing is not scalable. Once built, a supercomputer cannot easily be expanded, whereas, distributed computing is inherently scalable; to process more transactions or handle more data or calculate faster, '*simply*' connect more computers into the task. Due to this scalability, distributed computing is now the dominant approach for all but the most mathematically intensive data analytics tasks. Indeed, even in such realms, it can now rival supercomputing performance. For example, the Folding@home[13, 14] distributed computing project, used in the mathematical simulation of protein dynamics, was the first computer architecture of any kind to break the exaFLOPS (10^{18} floating-point operations per second) barrier. This was achieved by harnessing the power of 4.63 million CPU cores and almost 430 thousand GPUs located in simple PC and gaming machines distributed throughout the internet.

This truly impressive feat demonstrates the power brought by the scalability of distributed computing and is facilitated by two fundamental aspects: high-speed computer networks, and very low-cost computer hardware. Many technologies have emerged that leverage these two aspects so that distributed computing and distributed data analytics are now ubiquitous throughout our daily lives. Almost all human transactions such as buying groceries, booking a cinema ticket and paying one's taxes as well as many forms of human communication, e.g., social networking, often involve messages being passed around and analysed by various distributed computer systems.

As previously implied, distributed computing is not actually as *'simple'* as connecting more computers into the task. Multiple layers of software technologies cooperate to hide the complexities that enable even a novice user to transact on or retrieve information from one of the millions of computer servers connected to the internet, effectively presenting the distributed hardware and information resources as a single entity, i.e., the internet.

Some of the fundamental issues involved in scaling for distributed computing are:

- **Size scalability:** Are all the computers of the same hardware type and operating system (typical of Cloud Computing) or is a mix of hardware types and operating systems to be used (Jungle Computing [15])?
- **Geographical scalability:** How are the distributed devices to be connected? In a single location (e.g., Cloud computing), devices are connected via high-

speed LANs enabling traditional synchronous application programs to run on the distributed system without imparting too many input/output performance bottlenecks. On the other hand, in geographically dispersed systems, applications must be adapted to employ alternate approaches, such as non-blocking asynchronous message passing, in order to avoid the overall system grinding to an input/output bound halt. Further, data replication and data consistency become far more complicated problems in geographically distributed systems.

- **Administrative scalability:** Who owns the computers, a single organisation, co-operating organisations or arbitrary customers on a pay-as-you-go basis? When multiple organisations or individuals share the computer resources, how are privacy and billing to be managed?

- **Response time scalability:** Are the applications and jobs to be executed time-sensitive? For example, Massively Multiplayer Online Games (MMOs) require real-time responses to the gamers' button clicks or can jobs be executed in batch or perhaps some mixture of both? Notwithstanding the very high-speed data communication technologies now employed in today's internet, communication speed falls by orders of magnitude as the connection link extends out to the end user who, these days, will often be on a wireless connection and mobile. The emerging fields of Edge and Fog computing are approaches that focus on locating servers as close as possible to the end user in order to try to reduce the communication latency that becomes a major barrier for the more established Cloud and Jungle computing approaches.

An in-depth discussion of all aspects of distributed computing is given in [16]. An excellent review of modern distributed computing architectures is given in [17] which identifies the main configurations (Cloud, Jungle etc.) in use today and the problems they are aiming to solve. In general, these architectures help solve the issue of enabling multiple computers and their associated hardware resources to appear as a single computer. On a higher level of abstraction, the development of virtualization technologies (e.g. virtual machines, containers, and so forth) allow applications to adapt elastically on-demand. The adoption of service interfaces (e.g. REST architectures, and micro-services) enables complex problems to be broken down into smaller, repeatable tasks, which enhance scalability compared to the complexities of approaches like monolithic Service Oriented Architecture (SOA). Workflows and WFMS are yet a higher level of abstraction that attempt to make it easier to build useful applications by formalising this process.

1.3 Workflows

The workflow methodology provides a robust means of describing applications consisting of control and data dependencies along with the logical reasoning necessary for distributed execution. For wired networks, there has been a wide variety of successful workflow systems available for researchers to design, test and run scientific workflows [18–27]. A scientific workflow is a set of interrelated computational and data-handling tasks designed to achieve a specific goal. It is often used to automate processes which are frequently executed, or to formalize and standardize processes. A workflow may be used to define and run computational experiments or to conduct recurrent processes on observational, experimental and simulation data. Scripting languages and graphical notations may be used to represent tasks in a workflow, and the dependencies between them. Similarly, in the business domain, the Workflow Management Coalition (WfMC) [28] has delivered standards to a vast number of business workflow systems for over twenty years.

For microservice architectures [29] operating over fixed networks, the underlying TCP/IP backbone guarantees sufficiently stable service endpoints and connectivity to facilitate the construction and management of multi-service applications using centralized WFMS. Centralization enables several significant advantages for workflow construction and operation:

- Job step allocation and data/parameter passing through a centralized controller greatly simplifies workflow orchestration since the WFMS knows the addresses and interface descriptions of each service step.
- A centralized WFMS can enforce rules on service descriptions, via agreed-upon ontologies, so that they can be stored in searchable central registries, which greatly simplifies service discovery and matchmaking.
- For time-sensitive applications, a centralized worldview greatly simplifies the tasks of load-balancing, scaling, and optimisation when executing multiple workflows across multiple heterogeneous compute resources.

Nevertheless, in the current state-of-the-art, workflows must be specified imperatively. The discovery of workflow steps is not dynamic; the exact services to be used for each workflow step, along with the IP-addresses, connections and data locations, must be specified in advance. In addition, due to the continuing advancement of wireless communication technologies, as well as the continual price fall and miniaturisation of CPU and memory chips, many ordinary devices, collectively known as the Internet of Things (IoT), are now being endowed with compute and communication capability.

IoT devices and mobile handheld devices typically operate at the edge of networks where Peer-to-Peer (P2P) operations would be advantageous. However, for military environments, the Internet of Battlefield Things (IoBT) [30] and handheld/mobile devices must often operate in dynamic, low bandwidth, high latency, transient, environments such as mobile ad hoc networks (MANETs) [31] making centralized WFMS impractical if not impossible to implement. In addition, P2P workflow discovery and orchestration architectures [32] that rely on Distributed Hash Table (DHT) for object discovery are known to perform poorly when network fragmentation and high latency causes churn in the maintenance of the DHT overlay's underlying routing tables [33].

Nevertheless, the burgeoning volumes of smart devices and sensors available at the edge of networks open up the possibility for devices to carry out collaborative tasks without reliance upon centralized controllers. For example, multiple edge devices could collaborate to complete a data analytics task that a single edge device could not solve (perhaps due to device resource limitations or missing sensor data available on a sensor nearby) [34, 35]. Secondly, a decentralized multi-user chat application could enable users to discover and join chats without connecting to a central registry. In these scenarios, on-demand workflows capable of spontaneously discovering multiple distributed services without central control, or central registries, are essential. The resulting distributed pathways are complex and, in some cases, impossible to manage centrally because they are based on localized decisions and operate in extremely transient environments.

Consequently, in dynamic environments, a new class of workflow methodology is required—i.e., a workflow that operates in a decentralized manner and is capable of discovering and intelligently co-opting service resources in a cognitive manner to achieve a desired intent. By '*cognitive*', I mean a cooperating set of resources capable of deciding between themselves from local (decentralized) information and without centralized control, the best set of resources to participate in a particular workflow request. By '*intent*', I mean the ability of the distributed cognitive service architecture to recognise and possibly learn similarities between workflow and sub-workflow requests when considering such requests as commands. For example, a logistics company may use multiple workflows to deliver a part to a specific destination. Two workflows that source the part from different locations, via different assembly lines, but can deliver to the same endpoint should be considered similar in terms of their '*intent*'.

In a more general sense, a workflow can be considered a sequence of operational steps needed to complete a task. The workflow paradigm is not confined to computer science; many human activities can be considered workflows. For example, the steps required to sow a field or recover a harvest in farming and those carried out by factory workers to create a part can all be considered types of workflows that humans carry out.

One interesting aspect of human orientated workflows is that the human brain is more adaptable to changing situations and unexpected events than any workflow engine and can improvise and discover new ways to carry out procedures. For decentralized workflow operation, especially in unreliable network environments, the ability to adapt and improvise in a human-like manner would be highly advantageous. For this reason, this thesis investigates the possibility of creating a new workflow methodology that combines the state-of-the-art in computer workflows with the flexibility and potential for adaption and improvisation offered by a sub-field of Artificial Intelligence (AI), namely Vector Symbolic Architectures (VSAs), also known as Hyperdimensional computing.

1.4 Vector Symbolic Architectures.

A Vector Symbolic Architecture (VSA) is a form of brain-inspired computing for representing and manipulating data in a high-dimensional vector space. Unlike classical computing, which operates on bits through logical operations and the four arithmetic operations of addition, subtraction, multiplication and division, VSAs deal with hypervectors through three operations, superposition (a '*bundling*' or addition operator), permutation/multiplication (a '*binding*' operator) and a vector normalisation operator. Its distributed representation of information inherently makes the computing robust, scalable and requires less time and data for training and inference.

Various vector space domains have been used for VSA implementations, including Plate's real-number Holographic Reduced Representations (HRRs)[36], and Frequency Domain HRRs [37] and Kanerva's dense Binary Spatter Code (BSC) vectors. Nevertheless, all VSAs use recursive bundling operations to build hierarchical vector representations of objects from their sub-component vectors. For example, in VSA, a document can be represented as a single VSA vector by hierarchically bundling its sub-components starting from a set of unique letter vectors as follows; letter vectors -> word vectors -> sentence vectors -> chapters vectors -> document vector. All vectors have the same size at every level in a concept hierarchy. The resultant top-level VSA vector reflects its sub-components so that, at each level in the concept hierarchy, vectors are semantically comparable.

VSAs have been used extensively in natural language processing [38–40] and cognitive modeling [41, 42] because they are neurologically plausible and capable of supporting a large range of AI tasks including: *(a)* Semantic composition and matching, *(b)* Representing meaning and order, *(c)* Analogical mapping [43, 44], and *(d)* Logical reasoning. For example, in the book ”How to build a brain”, Eliasmith [45] uses HRRs as the basis for the Semantic Pointer Architecture Unified Network (SPAUN) a direct application of hyperdimensional computing that successfully demonstrates many of these tasks using a single VSA model. For a summary and explanation of the tasks SPAUN can complete (copy drawing, recognition/classification, reinforcement learning, serial working memory, counting, question answering, rapid variable creation and fluid reasoning) see [45, Table 7.1, page 252].

In Kleyko [46] BSCs are used for various cognitive tasks, including a very compact temporal pattern classifier small enough for implementation on low-end edge devices [46, Page 45]. The implementation used only 1kB of memory for several hundred training samples. A '*one-shot learning*' implementation of the Hierarchical Graph Neuron pattern classifier algorithm [47] which, compared to the original implementation, has reduced time complexity (due to the parallel matching capabilities of BSCs) and improved noise resistance (due to BCS's inherent noise immunity) [46, Page 61] and a distributed online learning anomaly detection task [46, Page 107].

Accordingly, most VSA architectures can support a similar set of cognitive tasks. However, one important difference between HRR and BSC is that BSC implementations are computationally more efficient in terms of both memory (real numbers for HRRs cf. binary bit strings for BSCs) and processing for implementations (HRRs use discrete-time Fourier transform (DTFT) for binding cf. bitwise XOR for BSCs).

Hence, this thesis investigates the potential of using VSA as the basis for a new decentralized workflow methodology capable of robust and flexible operation in unreliable networks.

1.5 Hypothesis

> A VSA based upon BSCs [48] can be used to define a rich and yet compact encoding that will enable highly efficient representations of multi-modal service descriptions, decentralized service and workflow discovery, and distributed workflow execution. Further, this scheme will provide semantic matchmaking capabilities that can facilitate reasoning on service descriptions and service compositions, or workflows.

Applying VSAs to fulfil such objectives requires several research contributions, listed below, that collectively make up the significant original contributions of this thesis.

Research contribution R1 - Self-describing, multi-modal semantic service objects:
I show how VSA BSCs can be used to build self-describing, multi-modal vector representations of services, that allow for semantic fuzzy matching during service discovery. For example, text and sound sample vectors could be combined to describe an alarm sensor service. Matching can also be performed on approximate values, for example we can ask, "*find an object like ABC that is near this location XY*", or "*find an object ABC that has at least N noodles*".

Research contribution R2 - Hierarchical VSA bundling:
I extend VSAs using a novel hierarchical vector binding and bundling scheme that maintains the ability to perform semantic matches and avoids the false activation and nullification of embedded sub-vectors that plague previous schemes. The scheme is capable of recursively bundling multiple levels of abstraction (workflow and sub-workflows/branches) to a practically unlimited depth. The

ability to scale is a major advantage since it is not unusual for workflows to require many hundreds of individual steps in today's microservice workflow architectures. The new encoding encapsulates all the information needed to control workflow discovery and execution without a central controller. It also provides a unique context for every workflow execution step, which has the potential to enable service agents to learn the meaning of other agents (and themselves) in a similar way to how word meanings are learnt in Natural Language Processing (NLP).

Research contribution R3 - Holographic dynamic message sizing:

Shows how the holographic properties of BSCs can be exploited to obtain further bandwidth savings based on message content (the number of sub-feature vectors encoded in a request, reply or service description).

Research contribution R4 - Bandwidth efficient distributed arbitration:

I have created a bandwidth-efficient mechanism for distributed arbitration of service selection during workflow discovery that uses a delayed response mechanism (inversely proportional to match quality) to arbitrate the best match, often without the need for weakly matching responders to reply at all.

Research contribution R5 - Cognitive workflow model:

My computational model shifts workflow task allocation from a master/slave '*push*' architecture to a Peer-to-Peer (P2P) '*reactive*' model in which intelligent cooperation between self-describing service entities occurs to complete a workflow task. This is enabled via a one-to-many communication style (broadcast/multicast) so that autonomous workflow agents can monitor workflow requests and decide for themselves whether they have capability to satisfy a request. Combined with (R4), this creates a truly peer-to-peer architecture capable of operating in transient MANET environments without a central point of control.

These contributions are integrated into the thesis narrative as follows:

- **Chapter 2:** Describes the requirements and vision of my sponsors, which is to create '*distributed, federated brain*', i.e., a system that can discover and execute distributed microservices in transient MANET environments without central control. I outline the project's overall structure and goals and then describe the specific goals of my sub-project area by way of an example scenario that my sponsors created. From this, I extract a firm set of requirements that become the target requirements used to drive the research presented in this thesis.

- **Chapter 3:** Surveys the state-of-the-art related work considering the strengths and weaknesses of relevant areas that might fulfil the requirements laid out in Chapter 2 including: workflows and WFMSs; existing decentralized service discovery mechanisms; and alternate approaches for the creation of semantic objects that could be applied to the description and discovery of service objects including, Symbolic AI, Ontologies and Artificial Neural Networks (ANNs). Finally I introduce VSAs, highlighting why they became my proposed candidate for creating 'cognitive' workflows.

- **Chapter 4:** Describes the mathematical and statistical properties of VSA BSCs and explains why BSCs were chosen over alternate VSA representations. It identifies the operations that can be used to create compound VSA objects representing collections of sub-features, i.e., service descriptions, and ordered collections of sub-features, i.e., workflows, and explains why there is a limit to this. Finally, it discusses the standard methods used to represent ordered sequences in VSAs giving examples of why these approaches break down when attempting to create multilevel recursive objects like those necessary for the representation of complex workflows.

- **Chapter 5:** Describes how to create (shallow) semantic vector representations of microservices and service requests using VSA methods (R1). When used for matching during service discovery, the vectors are insensitive to order and agnostic to unknown terms. It introduces a novel encoding method that enables approximate matching on scalar values and shows how to convert existing semantic vector objects to BSC, which improves matching and allows for multi-modal vector embeddings (R1). Finally, it describes a method to automate the encoding of existing JSON/XML service descriptions files into semantic BSC vectors.

- **Chapter 6:** Describes a new hierarchical VSA encoding scheme (R2) that can be used to scale linear workflows to a practically unlimited number of workflow steps. The ability to scale is a significant advantage since it is not unusual for workflows to require many hundreds of individual steps in today's microservice workflow architectures. The chapter details how the new encoding encapsulates all the information needed to control workflow discovery and execution without a central controller and without need to specify IP addresses. Finally, it describes an empirical experiment that tests the encoding's semantic matching capability on a 20k sub-vector workflow hierarchy.

- **Chapter 7:** Describes how to leverage the holographic properties of BSCs for the dynamic optimisation of workflow orchestration messages before they are transmitted to the network (R3). It derives a mathematical model that can be used to calculate the minimum vector size needed to differentiate between similar Service Vector (SV) object descriptions and presents empirical verification of the model.

- **Chapter 8:** Building from the theoretical, empirical evaluations and implementations described in chapters 5, 6 and 7, this chapter describes how I brought these different components together into a VSA platform that has been used to address the various use cases described in Chapter 9. The overall computational model (R5) used to enable decentralized services to self organise and cooperate to complete a workflow is first presented along with an analogy of how a group of humans might cooperate to complete a task. The key features of the layer are then described.

It then describes a *python* implementation of the platform giving step-by-step descriptions and flow diagrams for both the active requester and listener/responder. Details of how these are extended to encode and execute DAG workflows is also described (R2). A real-time network emulator was used to emulate the operation of multiple services in a MANET environment, which allowed services to be moved and brought in and out of service.

- **Chapter 9:** This chapter describes the experiments and outcomes taken from my published papers, which are used to evaluate the theory and platform/architecture presented in the previous chapters.
- **Chapter 10:** The conclusion revisits my research question pointing out the gaps I identified in the state-of-the-art. It then expands on the research contributions outlined above before finally considering directions for future work.

Chapter 2

Requirements

This research is sponsored by The International Technology Alliance in Distributed Analytics and Information Sciences (DAIS-ITA) or DAIS which is an alliance of several universities, industrial and government research laboratories from the United Kingdom (UK) and the United States of America (USA) [34][49]. Through discussions with my project leads and co-researchers and extracted from various DAIS published and unpublished documents, this chapter examines the DAIS research vision for the creation of a distributed federated '*brain*' (a system that can intelligently compose, discover and execute workflows consisting of multiple cooperating microservices without central control in dynamic MANET environments) from which I extract a firm set of requirements that are used to drive the research for this thesis.

2.1 DAIS-ITA high level goals

The DAIS-ITA envisions future military operations will occur in both the physical and cyberspace realms. They look to leverage the new capabilities that are becoming available due to continuing technological advancement for tactical advantage. Such as, increased bandwidth from 4G and 5G, low-cost phones, wearables, IoT, sensors, and so forth. They envision a system of interconnected heterogeneous devices, sensors, and, when available, back-end infrastructure, owned by multiple coalition partners that can operate as a seamless whole taking into account the various priorities, data security, policies, and trust constraints of each partner. They recognise, also, that forward operations will occur in highly contested urban environments where digital communication, especially wireless, will often be low bandwidth and transient due to the nature of the environment. For example, large buildings can block wireless signals and jamming devices are easily hidden in such cluttered surroundings. Hence, a key requirement is that such an interconnected system should continue to operate, by finding alternate 'paths' to complete workflow tasks (i.e., alternate devices, sensors, or processing power), in *'real-time'* and in the face of such transient environments, while simultaneously dealing with the challenges of a dynamically changing situation in which power, computation and connectivity may be severely constrained.

In Verma et al. [35], some of the DAIS technical leads lay out their motivation and vision for the creation of a "*distributed federated brain*", i.e., a distributed Cognitive Computing System (CCS). They recognise that when high speed reliable and inexpensive networks are available, adopting a centralized processing paradigm for a CCS has many advantages. However, they describe situations where network connectivity is often poor or even absent, including environments with mobile endpoints such as automobiles, ships, drones, trains and robotic mules that need to move over wide geographical areas. Furthermore, in common with the motivations behind fog computing [50] and mobile edge computing [51], they recognise that there are several conditions under which a centralized CCS may underperform compared to a distributed CCS, even where the network connectivity is favourable. For instance, when devices are generating a significant amount of data, extracting insights from the data can be computed more efficiently near the location of data generation, as opposed to moving the data to a central location. Also, as the processing power of distributed devices increases over time, decentralized cognitive solutions can become more responsive, scalable, and inexpensive.

In [35], the DAIS authors further identify that a 'distributed brain' must have several key properties. It must be self-healing and resilient since it has to operate in an environment where elements may lose connectivity to backend systems, and any of the small component systems may disappear in an unpredictable manner. It has to react rapidly to changing situations on the ground, so it must be predictive and proactive in the decisions it makes. To deal with a dynamic environment, the system must be self-configuring, agile and adaptive. Since it is dynamically assembled from a large number of independent components, it needs to be a cooperative and collaborative collective of individual components. Humans and machines have different types of analytic and cognitive capabilities. The 'distributed brain' must integrate human analytics capabilities into the machine analytics capability in a seamless manner.

Verma et al. [35] then identifies the following four attributes as key considerations that must be addressed in the attempt to develop a distributed federated brain:

- **Composability:** How do we compose smaller elements into a larger aggregate that works like a seamless whole? What are the principles that link the attributes of a component to the larger whole, and how can we compose components belonging to different organizations with partial visibility and control in an environment with limited resources?
- **Interactivity:** How do different computing elements and people interact with each other, both with other members of the groups and to external stimuli from the environment? How should we model and understand the interactions between different elements and information sources? How do different sub-brains work together as a larger aggregate brain?
- **Optimality:** How can elements work together to obtain the optimal results in an environment with constrained resources? How can analytics be performed so that optimal performance is obtained automatically, instead of requiring complex manual optimization?
- **Autonomy:** How can elements work together in a proactive manner understanding future situations sufficiently well to operate with a degree of autonomous

behavior? How can a system determine that autonomous operation is inappropriate and human intervention is needed? How can different elements simplify the cognitive burden involved to best assist humans in the loop when intervention is needed?

In the unpublished Initial Program Plan (IPP), these four attributes are then divided into the following six project areas:

- **P1: Software Defined Coalitions:** will explore the principles by which different elements across a coalition could be composed via control plane interactions to form a virtualized larger element.
- **P2: Generative Policy Models for Coalitions:** will investigate approaches for policy based management in a coalition environment with sufficient autonomy to its constituent elements.
- **P3: Agile Composition for Coalition Environments:** will explore new architectures in which analytics code and data of various types (ISR, HUMINT etc.) are mobile and composed together optimally.
- **P4: Evolution of Complex Adaptive Human Systems:** explores the properties of external groups relevant to a coalition operation, and how such external groups evolve over time.
- **P5: Instinctive Analytics in a Coalition Environment:** investigates approaches for data and services can be matched together to create the analytics services [i.e., workflows] that are autonomous and optimal.
- **P6: Anticipatory Situational Understanding for Coalitions:** explores new analytics algorithms for proactive situation understanding that can enable create intelligent advisors for human in the loop systems.

In the context of this thesis, a CCS architecture will consider the challenge from the perspective of an interacting network of cognitive micro-services. Our micro-services are cognitive in the sense that they comply with established principles of cognition such as those defined by the Core Cognitive Criteria (CCC) [8], which to a large extent incorporate the four key attributes of composability, interactivity, optimality and autonomy described above.

2.2 DAIS Cognitive Services Requirements

The research for this thesis comes specifically under the auspices of project **P5**. However, there is significant overlap with projects **P1** and **P3**. **P2** is also relevant since workflow composition must consider data and service security policies. The remaining part of this section, taken *verbatim* from the IPP, highlights a typical scenario along with some of the related challenges which the various project areas must cover. (References to *'we'* in this section should be read as *'The DAIS team'*):

> Tactical coalition networks have two unique challenges related to composability [of workflows]. First, control can be distributed across coalition members with different policies, priorities, resources, and levels of trust among each other. Coalition members require secure access to resources across coalition boundaries in order to interoperate and achieve desired mission goals. Second, the level of dynamism is very high, due to the variety and velocity of mobile resources (troops, vehicles, etc.), network partitioning, and incomplete information, among other factors. In particular, network fragmentation and re-integration can be a common occurrence and must be a central focus of any approach for composing resources. Coalition operations frequently require the creation of dynamic communities of interests to conduct operations. This requires establishing functional networks across organizations within minutes, if not seconds.

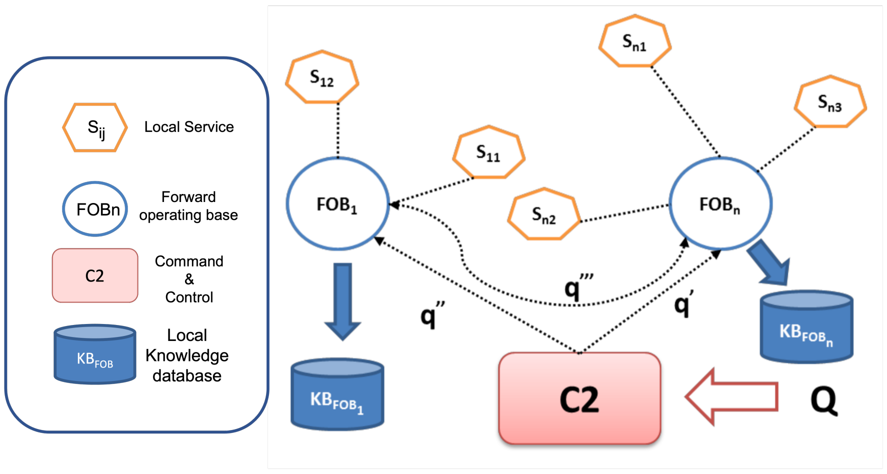


Figure 2.1: Distributed Analytics Scenario.

Consider the scenario depicted by fig. 2.1— the command and control (C2) obtains local (or global) situational understanding by querying information from forward operating bases (FOB), which in-turn get localized information from associated services (S_{ij}). These services [i.e., workflows] could be providing any information collection, information processing or information fusion functions. Furthermore, FOBs and $S_{ij}s$ may have communication links with varying quality to obtain localized situational awareness information from other resources, thereby improving their own situational understanding.

[DAIS believe that] future coalition networks will be composed of a set of heterogeneous assets, exposing their capabilities through micro-service architectures, that come together in an ad hoc manner to fulfil coalition needs, thus realizing a fully distributed information processing eco-system across the coalition boundaries.

However, one needs to have the means to bring these services together in a seamless and timely manner to support decision making in this new operational context; traditional approaches in which knowledge about services are centralized to match against user requirements will no longer scale in this setting. Thus, in this project, we require a system in which users specify their needs in a declarative manner, and the system infers required services (or compositions) by automatically discovering (or composing) them with respect to the declared user needs.

Project 5 provides a way to dynamically compose the analytics from the appropriate services and, together with Project 3, will provide insights into the development of a self-organizing declarative platform for analytics. This platform will underpin the analytics layer for the system that will perform information fusion for anticipatory situational understanding, addressed in Project 6.

In order to advance the current state of the art, we propose to undertake two research tasks.

- **Intelligent distributed analytic compositions:** We will explore approaches that can use semantic technologies to form and orchestrate services, and enable the attainment of a self-organising architecture.
- **Anticipatory instantiations of coalition analytics:** We will explore approaches and algorithms for optimally instantiating service instances.

In order to address the above-mentioned questions, we propose the following basic research activities:

- **Self-Describing Services:** We plan to explore new paradigms by which we can make services self-describing— as opposed to existing manual approaches—in order to enable seamless composition.
- **Self-Discovering Services:** We propose novel service discovery mechanisms wherein goals are declared and the system automatically finds the desired services—be they atomic or composite—on its own by matching the inferred features of services to the declared goals.

- **Self-Allocating Services:** We will introduce a rigorous framework for allocating services and resources to meet coalitional goals that accounts for different levels of specificity (resources might need to be allocated exactly as requested or sufficiently to fulfil a generalized intent).
- **Self-Provisioning Services:** In dynamic environments, it is important to have strategies to optimally reallocate or re-provision service plans. This is necessary, for example, when services unexpectedly fail, when more suitable services become available during plan execution, or when new exploitation opportunities of existing services are identified.
- **Non Von Neumann Analytics:** We propose to investigate the potential for performing distributed analytics based on brain-inspired computational models. We will explore the potential of neuromorphic processing to be extended to a distributed network setting, as a radically new approach for future coalition operations.

2.3 Summary

From Section 2.2 and the perspective of this thesis, the key objective can be summarized as follows; a means is required to discover and orchestrate sequences of micro-services, i.e., workflows, on-demand, using currently available distributed resources (compute devices, functional services, data and sensors) in spite of a poor quality (fragmented low bandwidth) network infrastructure and without central control. It is desirable to be able to compose such workflows *on-the-fly* in order to fulfil an *'intent'*. For example, during field operations, a soldier on the ground may request the identification of a vehicle as 'civilian' or 'military' using a command such as *'identify vehicle at location (x,y)'*. Is it possible to dynamically compose and orchestrate a workflow that will identify what local sensors and ANN classifier services are available in the area in order to fulfil such a request? Sensors and services, which are likely located on mobile devices, must be automatically discovered and connected to perform the analysis and return an answer to the soldier using the available wireless MANET connections and without centralized controllers.

From Verma et al. [35], in this approach micro-services are considered as semantic concepts and can be processed as such. It is in this sense a distributed CCS can truly be described as a ‘distributed federated brain’. The challenge is to develop an approach where micro-services are self-describing, can self-discover other micro-services (including data services, network services, policy and security services) and where micro-services are self-allocating and self-provisioning in the sense that they can optimally position themselves or be invoked within a network to perform the tasks demanded by users. To achieve these goals, we require a common way to represent our cognitive services and their capabilities. Distributed cognitive processing is then the patterns of information flow and influence that occur across the network. The resulting cognitive phenomena are a property of the larger systemic organization, rather than a property of the individual micro-services.

In summary, a set of firm requirements for this research task have been identified as follows:

- Investigate new methods that will enable services to be described semantically so that alternate candidates, capable of completing a specific workflow step, can be discovered in distributed network environments without the use of centralized repositories.
  - The ability to discover edge components, including sensors and functional data analytic capabilities (programs/services) located at the edge without central registers and domain name services, is needed.
- Investigate how workflows can be represented in a semantic manner that might

enable invocation of alternate similar workflows to fulfil an ‘intent’.

  - That is, can workflows be encoded so that they are semantically comparable, perhaps by considering and remembering the contexts in which workflows are executed or by using the order and alignment of similar individual workflow steps within a workflow in a similar way to comparing sentences in natural language processing?
  - In addition, a means of orchestrating workflows in a peer-to-peer manner is needed.

- Workflow operation must be robust to transient environments and capable of continuing/completing a task in the face of loss of connectivity and loss of participating actors (sensors/edge micro-services).
- We require the ability to operate in low bandwidth environments and to minimize the communications overhead required run workflows.
- It should be possible to implement such a new workflow/service discovery abstraction in non Von Neumann neuromorphic hardware architectures.
  - As noted in the requirements scenario, field operations are typically carried out using mobile devices. For this reason any approaches that can leverage the orders of magnitude reduction in power consumption promised by the emerging non Von Neumann neuromorphic hardware architectures is very attractive.

Chapter 3

Related Work

This chapter describes related work that is relevant to the thesis objectives. I survey the current state-of-the-art concerning workflows and WFMSs highlighting areas of weakness in the context of my requirements summary and specifically transient MANET environments. I then discuss the technologies available for service discovery and consider how these can be applied in distributed environments by assessing each approach's ability to perform fuzzy semantic matching. Alternate methods for creating semantic objects that could apply to the description and discovery of service objects are then discussed. Finally, I introduce VSAs, highlighting related work and why they are a candidate for creating 'cognitive' workflows.

3.1 Workflows and Workflow Management Systems

A WFMS is used to manage the complexity of executing the parallel processing streams of a distributed application, i.e., a workflow. It manages the various process steps that are required to execute a task on multiple machines, including the task partitioning, allocation and instantiation of the steps needed to complete the task and the collection and collation of the results returned from the parallel execution streams. A WFMS *workflow specification* describes how the distributed application is to be partitioned and the order of execution of the workflow steps, as well as the locations of the data resources needed for input and output. Some WFMSs use a programmatic style for workflow specification, for example, the Swift/T [52] and Pegasus [53] scientific WFMSs whereas others, such as Triana [54] [55] and Node-red [56], use service-based workflows and include a visual programming environment to enable black box compute/data nodes to be wired together graphically. The Apache Taverna WFMS [23] is a SOA based system that provides both a graphical workbench environment and scripting languages. The newest Taverna scripting language uses SCUFL2 [57], a formal ontology built on OWL WC3 semantic web standards, in order to enable support for service discovery [58], but service providers are centralized and require manual configuration.

For resource allocation and scheduling, modern WFMSs typically require the high bandwidth, stable network environments that are typical of LAN connected computer clusters and, for geographically distributed data analytics, the internet backbone. This allows the WFMS to maintain a worldview of the available resources from which it can employ complex resource allocation and scheduling algorithms to optimise the throughput of a shared set of computing resources. In addition, message passing between the partitioned sub-tasks always flows through the WFMS engine because, having scheduled and instantiated the tasks, it is the single point of control that is aware of the network location of each sub-task.

Hyperflow [27] is based on a formal model of computation called Process Networks, which uses asynchronous signals to coordinate flow. Such signals can operate in a decentralized way, but currently, there is no service discovery component; rather, it relies on the node.js [59] execution environment and employs third-party tools, such as *RabbitMQ* [60], to coordinate services. Petri net workflows [61] offer a decentralized approach by using directed bipartite graphs, in which the nodes represent transitions (*i.e.,* events that may occur, signified by bars) and places (*i.e.,* conditions, signified by circles). However, such workflows require predefined DAG-based workflows with concrete endpoints to be defined before deployment.

Newt [62] is designed to address network edge workflow environments by providing a reusable workflow methodology for decentralized workflows that incorporates decentralized execution and logic, support for group communication (one to many) and support for multiple transports *e.g.,* TCP, UDP, multicast, ZeroMq. However, although Newt has discovery interfaces available, it currently supports only pre-configured profiles for its nodes, so dynamic service discovery is not possible. In the Newt paper, the authors used the dialogue from William Shakespeare's Hamlet [63] as a workflow, where each actor is a node that decides what line to say and who to say it to, and the sending of those lines represents the network payloads. They argued that this example is highly illustrative of group conversations or distributed analytics at the edge, where complex local decisions are made and communicated to the distributed node(s) in a decentralized way. The play contains several instances where an actor speaks to several actors, thus creating natural distributed communications and there are other instances where an actor will speak to himself, causing looping.

The DENEB [64] business workflow system (based on a high-level type of Petri nets) supports runtime discovery of the service objects required to execute a business workflow and is an excellent implementation for mainstream internet e-commerce and business logic workflows since it can use formal methods to prove that the selected Petri net network will implement the desired business logic correctly. It also uses semantic web standards to facilitate service discovery. However, once again, the platform is designed around a set of manager middleware components that are unlikely to be effective in our transient, low bandwidth environments.

For IoT environments, more and more research is focusing on moving the data analytics task to the edge [65]. Emerging technologies such as Apache Edgent [66] and EdgeX Foundry [67] are aiming to define standards and programming platforms/architectures for edge devices that will enable data analytics at the edge. The primary objectives are to reduce the input/output and processing loads experienced by central servers as well as to reduce the consumption of communication bandwidth and to enable the possibility of real-time control of devices in control room scenarios such as factory assembly lines or large building management systems. For example, monitoring of a factory assembly line's sensors can be performed locally and in real-time on the edge devices until an anomaly is detected, which might then be relayed to a central server for further analysis and alarm/action processing. In contrast, consider the load imposed on a central server and the communication bandwidth cost of taking and processing thousands of factory sensor readings per second in real-time.

In all of these approaches, stable connectivity among edge devices and to the central server is assumed, as is a known network address location for each device. For more conventional data analytics environments, as previously stated, it is significantly more practical and efficient to maintain centralized catalogues with load balancing via a centralized system when high bandwidth reliable connections are available. In low bandwidth, highly transient MANET networks, such as those operating in military battlefield scenarios, it is impossible to rely on central registries because a single node can never be guaranteed to be available all of the time and consequently, a decentralized approach is needed. On-demand distributed analytics workflows for general collaborative environments need spontaneous discovery of multiple distributed services without central control [35]. Applying the current state-of-the-art workflow research to such dynamic environments is impractical, if not impossible, due to the difficulty in maintaining a stable endpoint for a service manager in the face of variable network connectivity; current workflow approaches are more focused on operating on highly available distributed computing infrastructures using TCP, centralized management and service discovery. Consequentially, for workflow architectures required to operate in highly dynamic MANET environments, service discovery is of critical importance and requires a different approach to the methods used when reliable communications is available.

3.2 Service Discovery

Service discovery is a difficult problem even when services are hosted in centralized repositories. Much research has been carried out for service discovery in MANET environments [68–73]. The additional complexity relates to service node mobility and associated frequent network fragmentation as well as limited bandwidth, latency and, for battery operated mobile devices, available power. An excellent survey of service discovery in MANET environments is found here [74]. The two main approaches used for service discovery in MANETs are directory based and directory-less architectures (hybrid approaches can switch between both mechanisms as network characteristics change). In proactive directory based protocols, services must advertise their presence to the network, via uni-cast (when dedicated directory servers are allocated) or broadcast/multicast. In reactive directory-less protocols, clients issue queries, usually via broadcast or multicast to which potentially multiple services respond. Each approach has advantages and disadvantages based on the MANET's operating characteristics. For example, in directory based approaches, directory maintenance imposes additional bandwidth and power demands on the network. If it is desirable to get whole network coverage for service advertisements and queries, directory-less approaches using broadcast or multicast can cause significant network congestion when there are high volumes of requests and service announcements in the network.

In both approaches, to facilitate matching, service announcements and client requests must be encoded in some form. Simple approaches use well defined/known service names or Universally Unique Identifiers (UUIDs) for example, Jini [68], mDNS [69–71]. For more complex matching, service attribute lists are used in the form of key-value pairs, e.g., INDI [72], ProtoSD [73]. Some architectures have attempted semantic web/ontology based matching [75–77]. However, in current approaches, as noted by [74] "it should be noted that in this class of protocols the ontology must be common for all nodes, and hence a priori agreed to among them".

A key requirement for DAIS is to locate objects that can fulfil the *'intent'* described by a workflow specification of a complex data analytics task in the face of very transient fragmented networks. If an exact service match cannot be found during workflow execution, finding whatever alternatives are available to complete the task is desirable. Thus, a mechanism that can perform semantic matching to discover objects in geographically local, highly fragmented networks is needed. This is complicated in coalition environments, where services are developed and deployed independently or by loosely cooperating partners in open environments, because the matching methods described here require search terms (UUIDs, attribute names, and ontology definitions) to be agreed upon in advance. For wireless military networks, a further issue with the current service discovery architectures is protocols requiring service announcements/advertisements may offer an adversary a chance to set up an attack on such a network. In the next section, we consider how services might be described semantically in order to obtain the most flexible and '*intelligent*' matching scheme.

3.3 Semantic representations and matchmaking

Knowledge representation [78–80] is an important sub-branch of AI and is used heavily by the cognitive sciences [43, 45]. This section considers the available approaches for semantic representation of complex objects, such as microservices, that will enable discovery of target services and alternatives during workflow execution. In addition, a means to semantically compare entire workflow and sub-workflow objects is required so that alternate workflows can be found and executed in response to the original workflows functional '*intent*'. For example, in the context of DAIS, a coalition member requesting a particular workflow to be executed could be fulfilled by a partner member but using the partner's version of the original workflow request. (Coalition members will cooperate to fulfil a goal but have their own rules, procedures and workflows.)

3.3.1 Symbolic AI

Symbolic AI uses '*atomic*' symbols to define higher-level concepts. The rules for manipulating and reasoning on such concepts are also defined using the underlying atomic symbols, as are the production rules (rules that define how to generate valid sequences of symbols). Examples of the symbolic approach to AI are, traditional programming languages, such as C++ and Java (which, via classes, use symbols to encapsulate data and the procedures that are allowed to operate on data), expert systems created out of traditional programming languages, cognitive modelling systems such as Adaptive Control of Thought-Rational (ACT-R) [81], and so forth. Traditional von Neumann, computers implement Symbolic AI systems. Computer ontologies are used to further facilitate complex semantic, search, retrieval and manipulation of objects and concepts in Symbolic AI approaches, particularly when such data are stored across the widely distributed heterogeneous computer storage and database sources that is the internet.

Ontologies

An ontology is a set of concept labels and the relationships between such labels used by both scientists and philosophers alike in attempting to 'externalise' and formalise the brain's inbuilt ability to understand objects and concepts in the world. In science an ontology tends to be specific and restricted to a specialized field. Many ontologies have been developed and published on subjects as diverse as physics, biomedical science, law, supply chain, colour, military operations and confectionery to name but a few. An ontology's main purpose is to represents knowledge, i.e., the attributes, properties and relationships of concepts and objects in a specific field. One aim of an ontology is to provide a common lexicon, that is, an agreed-upon set of symbolic tags, to enable individuals within and without organisations to communicate on a specialized subject. In computer science, an ontology enables machines to interface and communicate with humans as well as enabling autonomous machines to communicate and compare data via meta-data tags and well defined relationships. This approach has proven very successful for many knowledge-based systems and indeed much of the power of internet search engines comes from being able to process the meta-data tags embedded in HTML and XML documents. This is being extended further by Web Ontology Language 2 (OWL 2) and Web Ontology Language for Services (OWL-S) which are ontology languages aimed at further enhancing semantic lookup for internet linked data objects and are major components of “The Semantic Web” [82] as defined by The World Wide Web Consortium (W3C) [83]. Nevertheless, ontologies have some disadvantages, particularly when applied to the representation of distributed service objects.

Sabou [84], for example, provides a comprehensive review of issues relating to the use of OWL-S for web service representation and notes, “An investigation of OWL-S from an ontological perspective revealed that it presented conceptual ambiguity, poor axiomatization, loose design and narrow scope thus leading us to the conclusion that the ontology has a low clarity in semantics and these semantics are poorly formalized.” Additionally, [84] notes, “We observed that the generally available data sets for ontology learning are textual descriptions of the offered functionalities (e.g., API documentation, or Web service comments). These textual descriptions have a low grammatical quality and they employ natural language in a specific way.”

Dong et al. [85] surveys the state of the art with respect to Semantic Web Service (SWS) match-makers noting, “Unfortunately, none of the existing match-makers has focused on solving this issue by proposing a dynamic service discovery methodology that automatically adapts to different types of SWS [ontology] languages and parameters.”

With respect to the semantic web in general, McCool [86] observes, "The ontological data model makes representation of any nontrivial factual information difficult because it can‘t represent context of any kind." and "Each of these formats has seen phenomenally low adoption rates." leading him to declare in [87], "The Semantic Web has failed to produce communities, quantity, or quality."

Blass et al. [88] and Iliadis [89] describe one of the central issues relating to the use of ontologies in general, namely the Tower of Babel problem (ToB) which relates the problem of how domain experts can collaborate to achieve a common goal when they do not share a common language. In terms of ontology development, the ToB states that each time a new database or ontology is constructed, new terms are developed that represent an ever-changing language, thus complicating applied ontology-building, the goal of which is to produce semantically strong ontologies that can last over time [89].

In summary, when using ontologies for discovery of distributed service objects, some of the main issues are listed below.

1. Ontology specification/development requires a lot of effort and cooperation between domain experts and software engineers. Even then, ontologies only work best in narrow, highly specialized, use cases.
    - despite the effort represented by W3C and OWL 2, because of these difficulties, it is challenging to impose standards, especially in loosely cooperating coalition environments. Alatrish [90] compares five ontology editor tools and notably concludes "It is quite clear that Ontology development is mainly an ad-hoc approach".
    - In contrast, since its inception, the exponential growth of the internet is arguably due, in no small part, to the simplicity, ease of use and uptake of the original internet standard, HTML, URIs and HTTP [84].
2. Ontologies tend to become large and unwieldy and are difficult to extend and maintain [84, 91].
    - This is not ideal in an edge device scenario, particularly when considering maintenance and backwards compatibility.
3. Ontologies developed separately cannot be easily combined/merged.

- "The ToB problem is what prevents ontologists from realizing the full potential of ontologies. Each new category and relation from a different domain threatens to undermine an ontology by the heterogeneity of the labels and data structure." [89].

4. Concepts based on different ontologies cannot be easily combined (added together into a single concept) because semantic reasoning and comparison engines are specific to a particular ontology and do not work across separately developed ontologies without the use of complex translation/mapping techniques, which are often computationally expensive [85, 89, 92].

These issues are especially problematic when services are created in a coalition type environment where developers create services independently and often with little or no common standards. Further, Symbolic AI is considered to be notoriously brittle [93–95] because it is difficult for a programmer to account for every possible case / code path in a complex decision tree and such code paths cannot be easily adapted to changing situations. In addition, ontologies are difficult to perfect, extend and maintain. In contrast, the Connectionist approach to AI, i.e., ANNs are capable of adapting and 'learning' in changing situations. They are more capable at generalising, and can cope with imprecise inputs.

3.3.2 Connectionist, Sub-Symbolic AI

The Connectionist approach to AI [96–99] employs ANNs. These are large multilayered networks of mathematical units (artificial neurons), connected in matrices, that perform a relatively simple mathematical operation (such as integration) in parallel to discover statistical relationships in data of any kind including, image, sound, text, and speech. ANNs are very good at pattern matching and can often outperform humans at such tasks. They are capable of learning gameplay and machine control tasks. They can even exhibit emergent behaviour which, depending on the task at hand, may be desirable (e.g., new variations in gameplay) or undesirable (e.g., autonomous car piloting). ANNs are not good at complex reasoning tasks such as might be required in a situation where improvisation is necessary to get a successful outcome. Inspired by [100], some of the advantages and disadvantages of ANNs to distributed service discovery are highlighted below:

Advantages of Artificial Neural Networks (ANN)

1. **ANNs are distributed representations:** The set of weights associated with each neuron in a trained ANN (also known as a "model") represent where information is being stored about the data on which the ANN has been trained. Individual weights participate in the conversion of multiple individual data inputs to data outputs (classification) and multiple weights, but not all are effectively combined to create the output corresponding to a single input. (Note an "input" must also be presented to the network in some form of distributed set of values, i.e., a vector.)

2. **ANNs are robust to noise and corruption:** Distributed representations are robust to input noise and can continue to function even if part of the network is corrupted.

3. **ANNs are able to generalise:** For the same reason as 2), ANNs can produce sensible output for previously unseen inputs - this is the major power of ANNs.

4. **Parallel processing capability:** Due to the parallel organisation and execution plan of ANN models they are ideally suited for operation on GPUs (gaining significant speed-up), and they are also suited for implementation on Non von Neumann neuromorphic architectures (gaining speed-up and reduced power consumption).

Disadvantages of Artificial Neural Networks (ANN)

1. **Unexpected network behaviour:** The weights in a trained ANN are derived using statistical methods, which means that they represent a probability distribution and do not have 'defined' behaviour for a previously unseen input, i.e., they are non-deterministic. This makes them opaque to analysis and leads to the potential for unexpected outputs. Consequentially, this reduces trust in the network.

2. **ANN training:** ANN training takes considerable effort, often needing thousands of labeled samples and taking days or months to train even on dedicated highly parallel, expensive, hardware.

    - This is a particular problem when there are not many training samples available as is the case for the types of workflows used in military scenarios.

3. **Centralized architecture:** Due to point (2) as well as the sheer size of many modern ANNs, training must be carried out centrally. (Note, federated/distributed training of ANNs is currently in the early stages of research.)

4. **Fixed, Numeric Input Format:** The inputs to an ANN must be converted to a fixed size numerical vector format. Hence, an ANN cannot process a change in input format especially where additional input fields are required, i.e., they cannot easily cater for object descriptions of different size or shape (e.g., number of sub-features).

Hence, the Connectionist AI approach offers the potential that ANNs might be capable of learning the "meaning" of services, i.e., their transfer functions, and hence be capable of better generalising when trying to find an alternate service for some specific step in a workflow. In addition, the robustness offers potentially significant advantage in contested, transient MANET type environments. Also, ANNs are more neurally plausible, a DAIS requirement, and, since they are distributed representations, they might feasibly be implemented on the emerging Non von Neumann neuromorphic architectures (another DAIS requirement).

Nevertheless, the fact that ANNs are non-deterministic and need to be trained centrally with many training examples is detrimental. Further, it is difficult to see how workflow orchestration can be carried out using only neural networks. Consider the practically infinite number workflow graphs that could be created from a reasonable number of workflow steps (services). How could a neural network be trained to correctly orchestrate such a large number of graphs? Therefore, from the perspective of distributed service orchestration, it seems that the Symbolic AI approach may be more advantageous. (Currently all workflow orchestration engines are implemented in normal, symbolic, programming languages.) In addition, much work has been done in the service discovery area using Symbolic AI, i.e., the W3C, OWL 2, and OWL-S standards.

This dichotomy of advantages and disadvantages between the two approaches is highlighted, as a general issue, clearly by Kelly and West [101] along with a possible solution when they state, "The classic symbolic approaches to modelling do not account for how the symbol manipulations described in the model could arise from neural tissue, or account for how the symbols themselves come into existence. Classic connectionist approaches are more concerned with neural plausibility, but are notoriously opaque, doing little to aid our understanding of the cognitive processes modelled. By contrast, the **[vector-symbolic]** approach to modelling explicitly provides an account at both levels of description."

3.3.3 Vector Symbolic Architectures

Vector Symbolic Architectures (VSAs) [36, 46, 48, 102] are a family of bio-inspired methods for representing and manipulating concepts and their meanings in a high-dimensional vector space. They are a form of distributed representation that enables large volumes of data to be compressed into a fixed size feature vector in a way that captures associations and similarities as well as enabling semantic relationships between data to be built up. Such vector representations were originally proposed by Hinton [103] who identified that they have recursive binding properties that allow for higher level semantic vector representations to be formulated from, and in the same format as, their lower level semantic vector components. As such they are said to be semantically self-describing. Eliasmith coined the phrase *'semantic pointer'* [42, 45] to describe VSA *compound/concept* vectors because a VSA vector contains noisy copies of each sub-feature vector from which it was built, i.e., it is *'pointer'* to each sub-feature. That is, a high-level VSA concept vector can be *'unbound'* into a set of noisy sub-feature vectors each of which can be used to look-up the fully specified *'clean'* version of itself, stored elsewhere in *'clean-up memory'*. At the same time, the VSA compound vector simultaneously represents the collection of sub-features as a semantic concept (which can be manipulated directly without unpacking the sub-features).

Typically, in VSA, a basic set of symbols (e.g., letters in an alphabet) are each assigned fixed, randomly generated, hyper-dimensional vectors. Due to the high dimensionality, these vector symbols are uncorrelated to each other with a very high probability. Hence, they are said to be *atomic* vector symbols [48]. Vector *'superposition'* (a bundling operation) is then used to build new vectors that represent higher-level concepts (e.g., words) and these vectors, in turn, can be used to recursively build still higher-level concepts (sentences, paragraphs, chapters and so forth). These higher-level concept vectors can be compared for similarity using a suitable distance measure such as Normalised Hamming Distance (HD) or Cosine Similarity. VSAs have been used extensively in natural language processing [38–40] and cognitive modeling [41, 42] because they are neurologically plausible and capable of supporting a large range of AI tasks including: *(a)* Semantic composition and matching, *(b)* Representing meaning and order, *(c)* Analogical mapping [43, 44], and *(d)* Logical reasoning. Such properties are desirable to the challenge of creating cognitive workflows since they have the potential to fulfil many of the requirements listed in Section 2.3. The ability to represent both meaning and order in a single fixed-size representation suggests the ability to combine both semantic service object descriptions and ordered workflow steps into a single compact abstraction. Semantic matching and analogical mapping suggest the possibility of being able to find workflows that are encoded differently but that ultimately carry out the same, or a similar, function, i.e., fill an 'intent'. Since they are self-describing, they could be used to contain all the information needed to operate workflows without centralized control. Since the representation size does not grow as the complexity of the object (service or workflow) grows, they can provide a fixed size overhead for low bandwidth communication environments. Since VSA vectors act as both description and address of an object, they may be used to discover objects semantically as well for passing data between objects. (For example, by using the VSA vector as a parameterized address of an object in a multicast Remote Procedure Call (RPC).

3.4 Summary

In summary, the state-of-the-art workflow management systems use centralized controllers and are reliant on high bandwidth stable connections. Where this configuration is supported, it is highly effective. However, when connectivity is not guaranteed, it is essential that workflows can operate without central control and are able to find alternate services, especially when the workflow task is time-critical.

State-of-the-art WFMSs employ Symbolic AI and ontologies (W3C and OWL 2). However, Symbolic AI is brittle and inflexible when it comes to coding complex behaviours that need to adapt and change over time. Similarly, ontology creation and development become problematic as complexity and term coverage requirements increase. On the other hand, Connectionist AI is good at generalising upon many types of data. It is robust to noise, and the representation is compatible with emerging neuromorphic architectures. Nevertheless, training requires centralized hardware and large volumes of data. Also, ANN output to previously unseen data is non-deterministic.

A Vector Symbolic Architecture (VSA) has the potential to gain the benefits of both sides of the argument. At the symbolic level, VSA objects can be manipulated, combined and operated on in the same way as conventional Symbolic AI using well-defined mathematical operators. Since each symbol has an underlying distributed representation, like Connectionist AI, they are robust to noise. They can generalise since when VSA symbols are combined/bundled, the sub-components bear semantic similarity to the bundled object.

Chapter 4

Fundamentals of Vector Symbolic Architecture and Binary Spatter Codes

This chapter first considers types of VSA explaining why I chose to base this work on Binary Spatter Codes (BSCs). Details of the statistical properties of BSCs are then examined because they are important when assessing BSC vector comparisons. They also affect the number of sub-feature vectors that can be combined to create a concept (i.e., service description or workflow). I show how multiple BSC vectors are bundled together into compound objects and explain why there is a limit to this and, hence, to the ability to represent complex service and workflow objects without using recursion. A review of methods for creating 'set' like objects is given, showing how to access individual set members (i.e., sub-vectors). Finally, I discuss the standard methods used to represent ordered sequences in VSAs and give examples of why these approaches break down when attempting to create multilevel recursive objects like those necessary for the representation of complex workflows.

4.1 VSA types

VSAs use hyper-dimensional vector spaces in which the vectors can be real-valued, such as in Plate's HRRs [36], typically having dimension (D), where ($512 \leq D < 2048$), or they can be large binary vectors, such as Pentti Kanerva's BSCs [48], typically having $D \geq 10,000$. As mentioned in Section 3.3.3, semantic concepts are represented by combining multiple sub-feature vectors into a single concept vector using a bundling operation. The statistical variance and standard deviation of similarity comparisons is directly proportional to $1/\sqrt{D}$ (see Fig. 4.1 and Eq. 4.3 for BSCs and [36] for HRRs). Higher dimensionality, therefore, increases usable sub-vector 'memory capacity'.

For example, when sub-vectors are orthogonal, Kleyko [46, Paper B, page 80] estimates the capacity of 10kbit BSCs to be approximately 89 sub-vectors. In [36, Appendix C], Plate derives a lower bound for the capacity of HRR superposition memories when sub-vectors are orthogonal as

$$k \approx \frac{D}{3.16 * \ln(m/q^3)} + 0.25 \tag{4.1}$$

Where:

k = number of sub-vectors that can be successfully decoded.

D = the dimensionality of the vectors.

m = the the total number of vectors in the '*vocabulary*' or search space.

q = the probability of error in decoding all of the contained sub-vectors correctly.

When $D = 10,000$, $m = 100,000$ and $q = 10^{-6}$, equation (4.1) estimates the capacity of HRRs to be $k = 62$ sub-vectors which compares well to Kleyco's result of 89 for BSCs. In practice, however, BSCs typically use higher dimensionality that HRRs. This is because HRRs use real-number vector elements compared to bit field elements for BSCs. Therefore, BSC vectors consume only 1.56% per dimension of the memory consumed by an equivalent, 64bit float, HRR implementation. In addition, HRRs use circular convolution (discrete Fourier transforms) whereas BSCs use bitwise exclusive-or, a much simpler operator. Hence, BSCs have a significantly lower cost in terms of memory and processing for implementations. Indeed, Recchia et al. [40] compare circular convolution (HRRs) and random permutation (BSCs) for performance and memory capacity on a natural language task finding, "Random permutations outperformed convolution with respect to the number of paired associates that can be reliably stored in a single memory trace. Performance was equal on semantic tasks when using a small corpus, but random permutations were ultimately capable of achieving superior performance due to their higher scalability to large corpora." For the large corpus, [40] found the HRR solution to be intractable, reporting issues with computation time and storage capacity and noting the computational complexity of HRR solutions to be $O(k.log(k))$ compared to $O(k)$ for BSCs.

Complex workflows might consist of many hundreds of microservice steps requiring the ability to store many sub-feature vectors. Hence, taking into account usable memory capacity and CPU processing cost, 10kbit BSCs vectors[1] was chosen as a basis for the work here. Nevertheless, it is noted that most of the equations and operations discussed should be compatible with HRR solutions.

4.2 Binary Spatter Code Vector Space Distribution

BSCs of dimensionality 2^{10k} represent a vast vector space. By comparison, a commonly accepted estimate for the number of particles (protons, neutrons, neutrinos, electrons, and photons) in the known universe is only 2^{561} [104]! As mentioned in Section 3.3.3, typically in VSA, a basic set of symbols (e.g., letters in an alphabet) are assigned fixed, randomly generated vectors that are nearly orthogonal to each other. These are then combined to build new vectors that represent higher-level concepts (words, sentences, paragraphs, chapters and so forth) which can then be compared for

[1]From now on when using the term 'vector', unless otherwise stated, I mean Kanerva's dense BSC vectors.

similarity using HD or, since we are concerned with matching rather than differences, Normalised Hamming Similarity (HDS), i.e., $HDS = 1 - HD$.

When generating the *atomic* symbol set (e.g. alphabet vectors), the probability of any particular bit being set to a 1 or 0 is 0.5. Hence, for very large independent and identically distributed (I.I.D) random vectors, the result will be, approximately, an equal 50/50, split of 1s and 0s distributed in a random pattern across the vector. Therefore, when comparing any two such randomly generated vectors, the expected value will be $HD{=}HDS{=}0.5$.

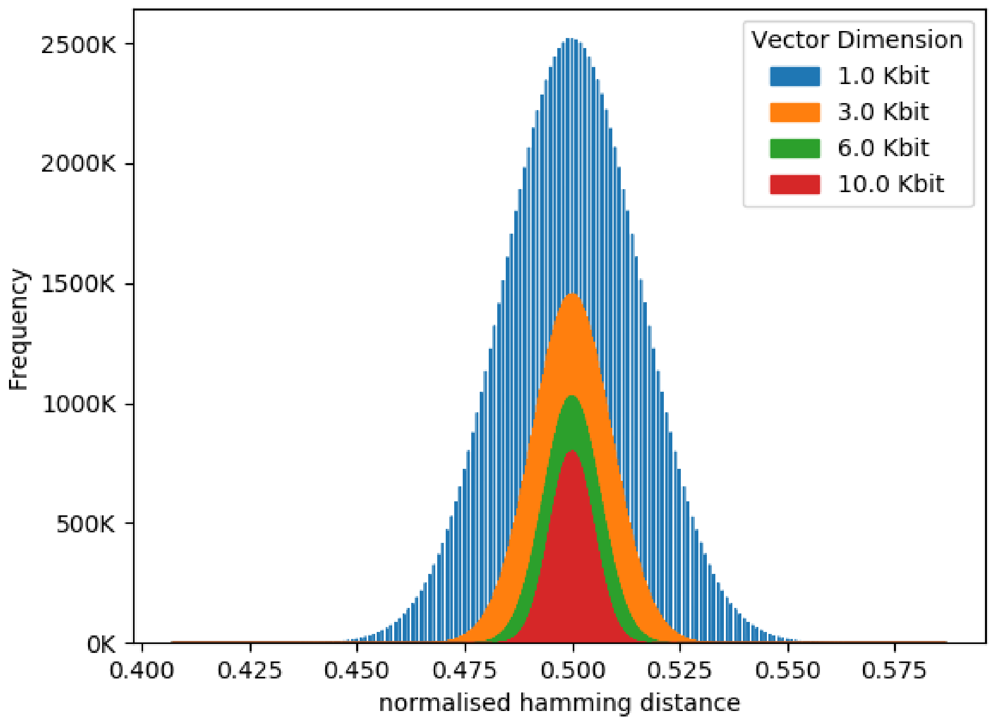


Figure 4.1: Distribution of random vectors with increasing dimension, 10^8 **samples for each vector size.**

The HDS comparison of two D dimensional, vectors is equivalent to carrying out *'D'* Bernoulli trials. Therefore, such comparisons will have a binomial distribution, which for a high number of comparisons is closely approximated by the normal distribution, see Fig 4.1. The variance, Eq. 4.2, and normalized standard deviation, Eq. 4.3 of the binomial distribution are shown below.

$$var(\mu) = D * p(1-p) \tag{4.2}$$
$$\sigma(\mu) = \sqrt{D * p(1-p)}$$
$$\sigma_n(\mu) = \sqrt{p * (1-p)/D} \tag{4.3}$$

Thus comparing two I.I.D random BSC vectors of length $D = 10kbit$ having bits $b_i \in \{b_0, b_1, \cdots, b_{D-1}\}$, where $P(b_i = 1|0) = p = 0.5$, gives,

$$var = 10000 * 0.5(1-0.5) = 2500$$
$$\sigma = \sqrt{2500} = 50$$
$$\sigma_n = 50/10000 = 0.005$$

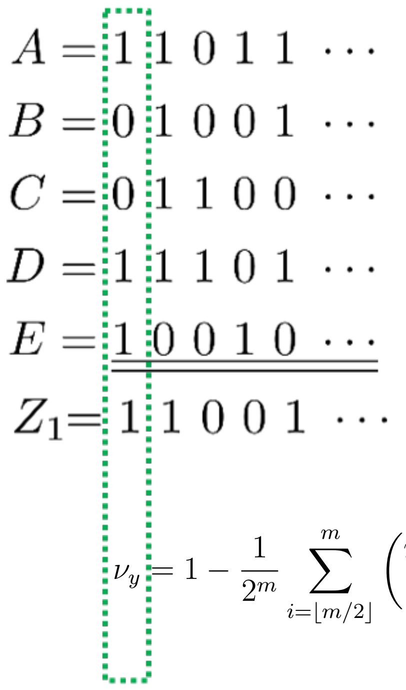
Due to this very tight variance, a practically unlimited number of *atomic* random vectors can be generated as needed, on the fly, without fear that the newly generated vector will be mathematically similar to any existing vector in the vector space. For example, fewer than 1 in 10^9 of any two such vectors will be closer in similarity than $HDS >= 0.53$, see Fig. 4.1; that is, further away from the HDS mean than six standard deviations, or differing in only 4700 bit positions instead of approximately 5000. Moreover, by implication, when comparing *'concept'* vectors a result of $hds(v1, v2) >= 0.53$ implies a significant similarity with a probability of error $\leq 10^{-9}$. A value of $hds(v1, v2) >= 0.524$ between two vectors implies a match with a probability of error of $\leq 10^{-6}$.

4.3 VSA Bundling operations

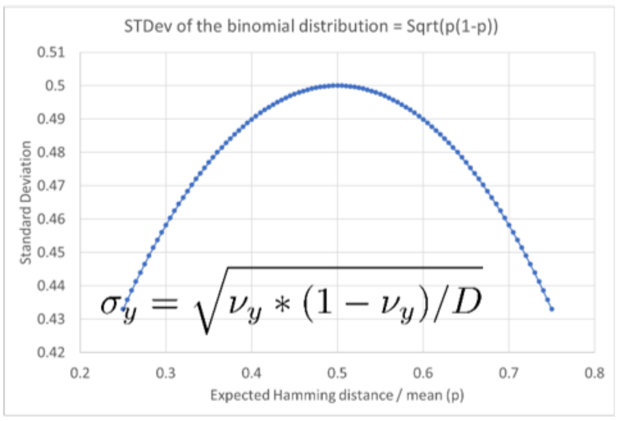
For BSCs, *bundling* is performed using bitwise *'majority_sum'* addition (i.e., bitwise majority voting) [48]. Simply put, for any particular column of bits in the sum, the majority wins; ties are broken randomly. In this text we will denote the majority sum operation of n vectors as $[V_1 + V_2 + \cdots + V_n]$, where $[\cdots]$ indicates the normalisation step. Hence, $Z = [V_1 + V_2 + \cdots + V_n]$ means add the sub-vectors first, then normalize so that, for any bit position i, set the corresponding output bit Z_i as follows,

$$Z[i] = \begin{cases} 1, & if \ (\sum_{j=1}^{n} V_j[i])/n > 0.5 \\ 0, & if \ (\sum_{j=1}^{n} V_j[i])/n < 0.5 \\ random, & if \ (\sum_{j=1}^{n} V_j[i])/n = 0.5 \end{cases} \tag{4.4}$$

The resulting vector, Z, is often referred to in the literature as a *'memory trace'* or alternately, *'compound vector'*, as well as *'concept vector'*. When the number of sub-vectors to be combined is greater that the decodable capacity of a single vector, the sub-vector list can be split into a number of smaller groups each of which is then bundled into *chunks*. These chunks can then be bundled into a single vector representing the entire concept in a hierarchical manner, a procedure known as *chunking*. The output of a bundling operation is of equal size to its sub-feature vectors and represents the lossy *superposition* of these components such that each vector element in the result participates in the representation of many entities, and each entity is represented collectively by many elements of the resultant vector [41]. The sub-vectors of a sum, such as Z above, can be probed for using HD or HDS. Since BSCs are I.I.D, the expected normalized hamming distance, μ_y, of such a test can be formulated by considering the majority sum over a single bit-column as shown in Figure 4.2. Consider the column outlined in green in the figure, from the point of view of each sub-vector, the bit in the sum vector $Z1$ will match that of the sub-vector if half or more of the other sub-vectors have the same bit value. For BSC vectors containing n, *distinct* sub-vectors, the mean expected value, μ_y, is given by Eq. 4.5. This is equivalent to the normalized hamming distance and is independent of vector dimension. The vector dimension, D, determines the variance of the mean and is given by eq. (4.3). In Figure 4.2, note that, for a given D, the standard deviation of the mean is greatest when vectors are orthogonal ($mean = 0.5$).



$A = 1\,1\,0\,1\,1\,\cdots$
$B = 0\,1\,0\,0\,1\,\cdots$
$C = 0\,1\,1\,0\,0\,\cdots$
$D = 1\,1\,1\,0\,1\,\cdots$
$E = 1\,0\,0\,1\,0\,\cdots$
$Z_1 = 1\,1\,0\,0\,1\,\cdots$



$$\nu_y = 1 - \frac{1}{2^m}\sum_{i=\lfloor m/2 \rfloor}^{m}\binom{m}{i} \quad \begin{cases} m = n & if\ n\ even. \\ m = n-1 & if\ n\ odd. \\ n = \#\ of\ vectors. \end{cases} \tag{4.5}$$

Figure 4.2: Expected, hamming distance, μ_y of a sub-vector memory-trace.

The *majority_sum* for odd numbers of vectors is always well defined since for each column, $count(0s) \neq count(1s)$. For even numbers of vectors, ties $'?'$ are broken randomly, for example,

$$\begin{array}{r} 10110101 \\ +\ \underline{00110011} \\ \underline{?0110??1} \end{array}$$

This can be achieved by appending a randomly generated vector into the sum. For bit columns already having a majority (marked as '⊠') the random vector does not affect the sum's output bit,

$$\begin{array}{r} 10110101 \\ +\ 00110011 \\ +\ \underline{0⊠⊠⊠⊠01⊠} \\ \underline{00110011} \end{array}$$

As mentioned above, for a given vector dimension, there is a limit to which sub-vectors can be bundled into a single chunk vector while remaining detectable. This should be fairly evident since, when increasingly higher numbers of sub-vectors are bundled together, the chances of any particular bit in the sum vector matching with the same bit in one of its component vectors approaches 50%. This is precisely the point at which the individual component vectors become undetectable within the sum vector. Figure 2 shows the effect of adding increasingly higher numbers of vectors together. Each point in the blue curve is the result of $hd(v_i, Sum_v)$ where v_i is each of the sub-vectors in the corresponding Sum_v. As can be seen, as more vectors are included into a single sum, the HD of each sub-vector approaches the orange curve centered on 0.5 which is the HD distribution obtained for comparison of multiple random vectors, i.e., the Sum_v vector becomes orthogonal to its sub-vectors.

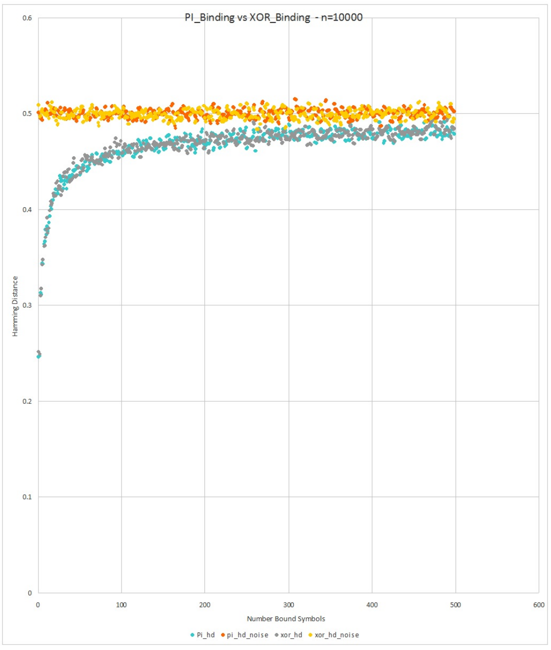


Figure 4.3: HD of sub-component vs Number of component vectors.

4.3.1 Bundling pitfalls of the Majority Sum.

In addition to the limited sub-vector storage capacity of BSCs, it is important to highlight out some other pitfalls related to the majority sum that must be avoided.

Addition of repeated terms

Addition of repeated terms when using majority rule addition is shown below: [2]

$$[A_v + A_v] = A_v \tag{4.6}$$

$$[A_v + B_v + A_v] = A_v \tag{4.7}$$

$$[A_v + B_v + A_v + C_v + A_v] = A_v \tag{4.8}$$

Whenever one particular vector term has a majority presence of one or more over the sum of alternate terms, then the sum collapses to equal that member's value. This is problematic since the non-dominant sub-feature vectors are lost from the resultant concept vector. This issue can be resolved using a permutation operator that alters repeating terms in a recoverable way to prevents such clashes. For example (using $'\cdot'$ to indicate bitwise exclusive-or), we could *XOR*[3] each term with unique positional *'role'* vectors ($p0_r, p1_r \cdots$) before bundling,

$$Z = [p0_r \cdot A_v + p1_r \cdot B_v + p2_r \cdot A_v + p3_r \cdot C_v + p4_r \cdot A_v] \tag{4.9}$$

The position vectors effectively permute or map the value vectors to a different part of the hyper-space making each permuted term unique. A simple, alternate, way of permuting BSCs is to perform a cyclic-shift on the vector [40]. This works because large vectors, having an I.I.D random distribution of 1s and 0s, look completely different if the bit-pattern is rotated by any number of steps less than the vector dimension. Using the exponentiation operator to indicate cyclic-shift[3], we can resolve the 'repeated term' issue by cyclically-shifting terms as shown below,

$$Z = [A_v^1 + B_v^2 + A_v^3 + C_v^4 + A_v^5] \tag{4.10}$$

[2]Remember that $[\cdots]$ indicates normalisation.

[3]In order to save space and aid readability in vector bundling/binding equations, we use '$\cdot$' to indicate the bitwise exclusive-or operator and the exponentiation operator to mean cyclic-shift ($+ve = shift\ right$, and $-ve = shift\ left$). The '$*$' symbol is used for ordinary multiplication.

Section 4.4 describes how these permutation schemes can be used to access individual sub-vectors within a bundled compound vector, and Section 4.5 describes how we can use them to encode ordered sequences.

Pairwise addition of terms

Pairwise addition produces an interesting effect, if for example, a number of vectors are added in a pairwise manner as shown:

$$Z_v = [\,[\,[\,[\,[A_v + B_v] + C_v] + D_v] + E_v] + F_v] \tag{4.11}$$

This method of addition acts as a short term memory such that only the most recent components are detectable in Z_v. In fact, the oldest components rapidly disappear from the memory trace Z_v as shown in the following output using 10kbit vectors:

$$\begin{aligned} Test\ sequence,\ Z_v &= [\,[\,[\,[\,[A_v + B_v] + C_v] + D_v] + E_v] + F_v] \\ hds(A_v,\ \ Z_v) &= 0.5161 \\ hds(B_v,\ \ Z_v) &= 0.5213 \\ hds(C_v,\ \ Z_v) &= 0.5345 \\ hds(D_v,\ \ Z_v) &= 0.5555 \\ hds(E_v,\ \ Z_v) &= 0.6277 \\ hds(F_v,\ \ Z_v) &= 0.749 \end{aligned}$$

After just five additions A_v and B_v are considered undetectable since their hamming similarity is below 0.53. The rate at which older sub-features vectors disappear from the *memory trace* can be controlled to an extent using a weighted or probabilistic add algorithm. However, it is better to perform bundling/majority sum operations as *one-shot* operations, alternately referred to as *late-bundling*. When the sub-vectors needed to create a compound vector are created serially or in a piecemeal fashion, it is better to maintain an un-normalized chunk vector and a sub-vector count for later normalisation than performing pairwise bundling and normalisation.

4.3.2 Using VSA to represent sets

If two high level concept vectors contain a number of similar sub-features, such vectors are said to be *semantically* similar. For example, we can create compound objects analogous to mathematical *sets* as follows:

$$\begin{aligned} Person1_v &= [\ John_v + Charles_v + 55yrs_v + T2Diabetic_v\ ] \\ Person2_v &= [\ Lucy_v + Charles_v + 55yrs_v + T2Diabetic_v\ ] \\ Person3_v &= [\ Greg_v + Charles_v + 34yrs_v + T2Diabetic_v\ ] \end{aligned}$$

HDS can be used to compare such vectors *without* unpacking or decoding the sub-features. Using HDS to compare $Person1_v$[4] with $Person2_v$ will give a match since they have 3 common sub-features. Also, $Person1_v$ and $Person2_v$ are more similar to each other than they are to $Person3_v$. An issue arises, however, when using VSA superposition to represent sub-feature collections as unordered sets. Consider the following record for example,

$$Person4_v = [\ Charles_v + Smith_v + 55yrs_v + T2Diabetic_v\ ]$$

$Person4_v$ is equally similar to $Person1_v$ as it is $Person2_v$ despite the obvious difference in the record.

4.4 VSA Binding operations

In order to resolve such issues, VSAs employ a *binding* operator that allows vector values such as $Charles_v$ and $55years_v$ to be associated with a particular *key*, *field name*, or *role*, within the data structure. Here we are using *field name* in the conventional sense used for data structures—i.e., it is the name of a subfield within a data structure.

[4]Throughout this text, a symbol having suffix *v* (X_v) depicts a vector that represents a value; a symbol having suffix *r* (Y_r) represents a known *atomic*, *unique*, *role* vector.

Role is an alternate description of the same and is more easily understood as a conventional variable name. For example, the variable *deposit_amount* might play the role of dollars being deposited in a banking transaction program.

An *atomic role vector*, or simply *role vector*, is an I.I.D randomly generated vector that is unique and nearly orthogonal to all other vectors in the vector space for the reasons explained in Section 4.2. When an atomic *role* vector is bound to a vector *'value'* this results in a *role-filler* pair which is analogous to variable assignment in conventional programming. For example, the statement $deposit\_amount = 300$ is said to *bind* the value 300 to the variable $deposit\_amount$. In a similar way, feature values such as $Charles_v$ can be bound to a role vector and detected or extracted from the *role-filler* pair vector using an inverse binding operator. Bitwise XOR is used for both *binding* and *unbinding* with BSCs because it is commutative and distributive over superposition as well as being invertible [48, page 147]. This means that both *roles* and *fillers* can be retrieved from a *role-filler* pair without any loss. For example, if $Z = X \cdot A$ then $X \cdot Z = X \cdot (X \cdot A) = X \cdot X \cdot A = A$ since $X \cdot X = \hat{0}$ (i.e., the zero vector) where '$\cdot$' represents the bitwise XOR operator. Similarly, $A \cdot Z = X$.

Due to the distributive property, the same method can be used to test for sub-feature vectors embedded in a compound vector as follows:

$$Z = [X \cdot A + Y \cdot B] \tag{4.12}$$

$$X \cdot Z = X \cdot [X \cdot A + Y \cdot B] = [X \cdot X \cdot A + X \cdot Y \cdot B] \tag{4.13}$$

$$X \cdot Z = [A + X \cdot Y \cdot B] \tag{4.14}$$

Examination of eq. (4.14) reveals that vector 'A' has been 'exposed', thus, if we perform $hds(X \cdot Z,\ A)$, we will get a match. The second term $X \cdot Y \cdot B$ is considered noise because $X \cdot Y \cdot B$ is not in our known *vocabulary* of vector features or symbols.

When a *role* vector and vector *value* are bound together using XOR, this is equivalent to performing a mapping or *permutation* of a vector's value elements within the hyperdimensional space so that the new vector produced is uncorrelated to both the role and filler vectors, for example, if $V = R \cdot A$ and $W = R \cdot B$ then R, A and B will have no similarity to V or W. However, comparing V with W will produce the same match value as comparing A with B. In other words, if A is closely similar to B then V will be closely similar to W because *binding* preserves distance within the hyper-dimensional space [48, page 147].

We note that *binding* with *atomic* role vectors can be used as a method of *hiding* and *separating* values within a compound vector whilst maintaining the comparability between compound vectors. This is an important property and can be used to encode position and temporal information about sub-feature vectors within a compound vector. It also explains why we can state that $X \cdot Y \cdot B$ from eq. (4.14) above will not match to any known symbol; however, note that we can get back to B from $X \cdot Y \cdot B$ by simply performing the appropriate XOR—i.e., $B \approx [[A + X \cdot Y \cdot B] \cdot X] \cdot Y$. We can now rephrase our *person* record in order to differentiate sub-features within the record by using a *role-filler* binding for each term; for example, we can formulate $Person1_v$ as,

$$Person1_v = FN_r \cdot John_v + SN_r \cdot Charles_v + Age_r \cdot 55years_v + Health_r \cdot T2Diabetic_v$$

This clearly resolves the incorrect matching between $Person1_v$ and $Person2_v$ with $Person4_v$. To test $Person1_v$ for the surname $Charles_v$ we perform,

$$hds(SN_r \cdot \ Person1_v,\ Charles_v) \tag{4.15}$$

For 10kbit vectors, if the result of eq. (4.15) is greater than 0.53 then the probability of $Charles_v$ being detected in error is less than 1 in 10^9 [48, page 143]. If our *person* record is distributed over a network we could transmit or multicast the request vector $Z = SN_r \cdot Charles_v + Age_r \cdot 55years_v$ to the network. Any listening distributed micro-service having person records containing the surname $Charles_v$ and age $55years_v$ can check for a match and respond or become activated.

4.5 Encoding Ordered Sequences

As mentioned in Section 4.3.1, a VSA can be used to encode an ordered sequence such as, A →B →C →D →E. In the context of a workflow composition, we consider that each letter symbolizes the vector representation of a specific type of microservice listening for work. That is, each letter $\{A, B, C \cdots\}$ should be considered to represent a compound/concept vector description of each microservice step that needs to be discovered to execute the workflow. These vector descriptions are built up using *role-filler* binding, *bundling* and, if necessary, recursive embedding of the lower level symbolic vectors that represent the sub-features of each microservices as described in Section 5.1. Three commonly used schemes for the representation of linear ordered sequences like the one above are shown below:

Cyclic-shift Scheme:

$$Z = [A^1 + \ B^2 + \ C^3 + D^4 + \ E^5] \tag{4.16}$$

Permutation Scheme:

$$Z = [p0_r \cdot A + p1_r \cdot B + p2_r \cdot C + p3_r \cdot D + p4_r \cdot E] \tag{4.17}$$

Self binding Scheme:

$$Z = [A + \ A^1.B + \ A^1.B^1.C + A^1.B^1.C^1.D + A^1.B^1.C^1.D^1.E] \tag{4.18}$$

To execute the sequence/workflow shown in each equation, the active requester must *unbind* the encoded steps sequentially. For example, to activate the first step, A, the cyclic-shift encoding (4.16) simply shifts Z one step left. For the permutation vector workflow (4.17) we XOR the workflow vector with $p0_r$. For the 'self binding' encoding (4.18) the first step is already exposed. The equations below show the first and second unbinding of each scheme.

Cyclic-shift Scheme:

$$Z^{-1} = [A + \ B^1 + \ C^2 + D^3 + \ E^4] \tag{4.19}$$

$$(Z^{-1})^{-1} = [A^{-1} + \ B + \ C^1 + D^2 + \ E^3] \tag{4.20}$$

Permutation Scheme:

$$\begin{aligned} p0_r \cdot Z &= [A + p0_r \cdot p1_r \cdot B + p0_r \cdot p2_r \cdot C \\ &\quad + p0_r \cdot p3_r \cdot D + p0_r \cdot p4_r \cdot E] \end{aligned} \tag{4.21}$$

$$\begin{aligned} p0_r.p1_r(p0_r \cdot Z) &= [p1_r \cdot A + B + p1_r \cdot p2_r \cdot C \\ &\quad + p1_r \cdot p3_r \cdot D + p1_r \cdot p4_r \cdot E] \end{aligned} \tag{4.22}$$

Self binding Scheme:

$$Z = [A + \ A^1.B + \ A^1.B^1.C + A^1.B^1.C^1.D + A^1.B^1.C^1.D^1.E] \tag{4.23}$$

$$A^1.Z = [A^1.A + \ B + \ B^1.C + B^1.C^1.D + B^1.C^1.D^1.E] \tag{4.24}$$

Some observations:

- **Cyclic-shift scheme:** Unbinding to activate the next step is very easy, simply repeat the cyclic-shift-left.
- **Permutation Scheme:** Activating the next step is complicated by the fact that a different permutation vector combination must be used for each unbinding, (first step is $p0_r$, second step is $p0_r.p1_r$, third step is $p1_r.p2_r, \cdots$).
- **Self binding scheme:** The activated service simply $XORs$ with a shifted copy of itself.
- **No position information:** If we consider the encoded sequences to be distributed workflows in all schemes, an activated workflow step (microservice) cannot tell in what position it has been activated.
- **Stopping criterion:** There is no obvious stopping criterion. If such workflows are being executed in a P2P distributed manner, how can the last step, service step E, know NOT to try an unbind and connect to the next service?

In the next section the disadvantages of equations, 4.16, 4.17, and 4.18 is described when they are used for recursive embedding.

Disadvantages of Cyclic-shift scheme

This type of sequencing is beautiful in its simplicity; however, it does suffer from some significant issues when trying to build more complex workflows. One major limitation is that it cannot support recursive embedding properly. Consider three word vectors built from alphabet vectors,

$$\begin{aligned}
Z_1 &= CRATER \rightarrow & Z_1 &= C^1 + R^2 + A^3 + T^4 + E^5 + R^6 \\
Z_2 &= ROT \quad\;\; \rightarrow & Z_2 &= R^1 + O^2 + T^3 \\
Z_3 &= ATE \quad\;\; \rightarrow & Z_3 &= A^1 + T^2 + E^3
\end{aligned}$$

As described above, the individual letters in each word are an analogue for three different workflows. Each letter represents an individual workflow step to be executed. (Each letter could represent a complex sub-workflow itself.) Then, as unbinding of Z_1 proceeds, false activations can occur. For example, Z_1^{-1} will activate Z_2 in preference to C^0 and Z_1^{-2} will activate Z_3 in preference to R^0.

$$\begin{aligned}
Z_1^{-1} &= C^0 + R^1 + A^2 + T^3 + E^4 + R^5 \\
Z_2 &= (R^1 + O^2 + T^3) \text{ matches 2 chars} \\
Z_1^{-2} &= C^{-1} + R^0 + A^1 + T^2 + E^3 + R^4 \\
Z_3 &= (A^1 + T^2 + E^3) \text{ matches 3 chars}
\end{aligned}$$

Disadvantages of Permutation Scheme

Again this scheme cannot support recursive bundling properly. A particular issue is how to maintain separation at multiple levels of nesting. Use of the same p-vectors for recursive bundling will cause false activation as shown in the example below:

Let

$$Z_1 = p_1 \cdot A + p_2 \cdot X$$
$$Z_2 = p_1 \cdot Y + p_2 \cdot A$$
$$Z_3 = p_1 \cdot A + p_2 \cdot Y$$

Create a higher level workflow consisting of steps Z_1 followed by Z_2

$$S_1 = p_1 \cdot Z_1 + p_2 \cdot Z_2$$

To activate first step in S_1, unbind with p_1,

$$\begin{aligned} p_1 \cdot S_1 &= Z_1 + p_1 \cdot p_2(p_1 \cdot Y + p_2 \cdot A) \\ &= Z_1 + (p2 \cdot Y + p_1 \cdot A) \\ &= Z_1 + Z_3 \quad \text{incorrectly creates and exposes } Z_3 \end{aligned}$$

One solution to this might be to use multiple sets of permutation vectors, one set per nesting level. However, what happens if we want to bundle a ‘p-level2’ object with a base level object? For example, let $Z_1 = p_{11}.A + p_{12}.B$. If we now want to create a workflow $Z_2 = X \rightarrow Z1_1$ what p-vectors should we use? If we know that Z_1 contains a ‘level-1’ sequence then perhaps we bind it as $Z_2 = p_{21}.X + p_{22}.Z_1$, but what if Z_1 is already a p2-level object, e.g., $Z_1 = p_{21}.A + p_{22}.B$? This would cause B to become activated incorrectly in a similar way to the problems encountered when attempting to use only one set of p-vectors. Further, determining the correct p-vector combination required to unbind the next workflow sequence step becomes significantly more complex.

Disadvantages of Self binding Scheme

Semantic matching does not work well; for example, consider the following encodings where each letter is regarded as a vector description of a service

$$CAT = C + C^1.A + C^1.A^1.T$$
$$CUT = C + C^1.U + C^1.U^1.T$$

Inspecting terms, clearly

$$C^1.A \neq C^1.U \quad \text{and} \; C^1.A^1.T \neq C^1.U^1.T$$

Longer workflows represented by words such as MAXIMUM and MINIMUM will differ in every position after the first 'M', hence possibilities for online matching and learning of sub workflows is inhibited.

4.6 Summary

This chapter shows that a VSA based upon BSCs has the potential to be used for the representation of service descriptions via *binding* and *bundling* as well as for the definition of workflows which can be encoded as ordered sequences. However, the limits to which sub-vectors can be bundled into a single BSC vector require an improvement on the current methods used to perform recursive vector embeddings that must avoid false activation and maintain the ability to perform matching. How these methods can be used and extended to create semantic service objects descriptions and workflows that can operate without a central point is described in the next chapter.

Chapter 5

Service description using a Vector Symbolic Architecture

This chapter details how microservices can become self-describing (R1) by creating (shallow) semantic SV representations of themselves using VSA encoding methods. These SV self-descriptions are used for matching to workflow requests. I introduce a novel VSA approach that extends this encoding to include approximate matching on scalar values (e.g., find an X of about *'this'* size). I also describe a method to improve the encoding's semantics and enable multi-modal vector embeddings. For example, using this method, vector representations of sound samples or images can be embedded directly into a service agent's SV description (that can be matched to similar samples in a workflow request vector). The method can also convert existing semantic word databases to BSCs for use when building SV descriptions from textual and JSON/XML service descriptions.

5.1 Semantic vector representations of services and QoS

Having shown how BSC can be used to represent data structures, I now consider how to represent service descriptions and their associated Quality of Service (QoS) as symbolic vectors (R1). In the following encoding scheme, recall that X_r represents a unique *atomic* role vector and Y_v represents a value vector and that $X_r \cdot Y_v$ is a role-filler pair that binds the category X_r to the filler value Y_v and enables later matching and retrieval of values by specific categories. I also note that it is perfectly valid to XOR *role* vectors together to represent sub-features as appropriate. Services build representations of themselves by bundling role-filler pairs together to create a compound Service Vector (SV) that can be matched upon during workflow service discovery. The SV is built following the base scheme highlighted by Eq. 5.1.

$$Z = Serv_r \cdot Serv_v + Resource_r \cdot ResP_v + QoS_r \cdot QoS_v \tag{5.1}$$

Where

- Z is the composite service description vector;
- $Serv_r \cdot Serv_v$ is the vector representation of the functional description of the service;
- $Resource_r \cdot ResP_v$ is a vector embedded into a request that points to any needed external resources. This is not part of a service's self-description but allows a matching service to locate any external resources specified by a requester;
- $QoS_r \cdot QoS_v$ is a vector representing either the requester's QoS requirements or the current QoS value for a specific service.

Building the description vector $Serv_v$

$Serv_v$ is itself comprised of symbolic vectors that semantically describe the essential elements of a service, in terms of role-filler pairs that are needed to find a match. To illustrate how this is achieved, an example of relatively simple service description comprising service name, inputs, outputs, and a functional description of the service is shown in eq. (5.2):

$$Serv_v = Inputs_r \cdot Inp_v + Name_r \cdot Name_v + Desc_r \cdot Desc_v + Outputs_r \cdot Out_v \tag{5.2}$$

Where

- $Inputs_r \cdot Inp_v$ describes the required inputs;
- $Name_r \cdot Name_v$ a vector encoding of the service name;
- $Desc_r \cdot Desc_v$ a vector encoding of the service's description;
- $Outputs_r \cdot Out_v$ describes the required outputs.

The filler/value components of these vectors are also built from symbolic vectors. For example, if our service, say Z, has three float inputs and one BitMap input then Inp_v can be encoded as:

$$Inp_v = One_r \cdot Float_r + Two_r \cdot Float_r + Three_r \cdot Float_r + One_r \cdot BitMap_r \tag{5.3}$$

One_r, Two_r, $Three_r$ are atomic role vectors representing numbers. Since data types are unique, $Float_r$ and $BitMap_r$ are also represented by *role* vectors. This simple scheme is adequate for representing input/output descriptions because microservices typically do not have high numbers of input/outputs. More complex input/output descriptions can be encoded via embedding further role-filler pairs. The above vector is a bag representing the inputs that enables flexible matching. If the input part of a request vector is encoded as:

$$InpReq_v = One_r \cdot Float_r + One_r \cdot BitMap_r$$

then the input description for service Z would constitute a match, and provided that the other sub-features matched sufficiently, including its semantically encoded QoS_v the service could become activated. Note that a different service having exactly one float and bitmap input would better match the input specification.

5.2 Representing Scalars and QoS

When encoding the sub-features of a concept (e.g., a service description), we will often need to encode scalar values, particularly when encoding QoS values. There are a number of ways to represent scalars depending upon the matching requirements of such values. For example, when a scalar is used to represent positional information, assign unique role vectors to each number in the expected range (e.g., One_r, Two_r, $Three_r \cdots$, as described above).

For QoS metrics, it is often required to meet a specific maximum or minimum value for the metric in question. For example, a static QoS metric could specify that a service must have at least 4 CPU cores available to fulfil a workflow request step. This can be treated as a request for a scalar range by creating a '*set*' of acceptable *'number role'* vectors and, via the distributive property of BSCs, associating such a range with the object property being requested, as shown in Eq. 5.4,

$$Reqired\_CpuCores_v = CpuCores_v \cdot [Four_r + Five_r + Six_r + \cdots + MaxCores_r] \tag{5.4}$$

Dynamic QoS metrics such as percent Battery life or *available runtime* can also be encoded in this way. For example, aggregate bandwidth, obtained by each service actively monitoring its local bandwidth with pings, could be quantized to $(1kb, 10kb, 100kb, 1Mb, 10Mb, 100Mb, 1Gb)/Sec$. Such values are converted to an enumeration, thereby allowing us to encode ranges representing different underlying values with the same role vectors. Thus, encoding a minimum Bandwidth QoS requirement of, say, 100Mb/sec would be encoded as follows:

$$Bandwidth_r \cdot [Six_r + Seven_r] \tag{5.5}$$

Integers (up to the vector dimension $D-1$) can be represented very simply using cyclic-shift, for example to encode any integer, i, as an *atomic* role vector cyclic shifted by 'i', i.e., $NumberBase_r^i$. Passing numeric metadata parameters like this makes decoding very easy. Simply loop, counting the number of '*shifts*' until a match is found between the received vector and $NumberBase_r$.

5.2.1 Approximate numeric matching (R1)

To match on scalar values that are numerically near to each other (for example, we might want to find an object that is nearest to a certain position), we can create a set of role vectors that represent a *'number-line'*, i.e., a sequence of vectors such that the hamming distance/similarity between each vector in the *number-line set* proportionally reflects the distance between steps in a range of numbers. Listing 5.1 describes how to generate a *number-line* set. Since BSCs have a fixed dimension, if the required number range is larger than the vector dimension, then the range should be normalized. The *orth_vec* parameter is used as a role vector that enables multiple *number-line* sets to be generated, each of which is orthogonal to all other sets. Starting with a unique role vector, *start_vec*, the algorithm flips bits repeatedly ensuring that the same bits are not flipped twice. Since number-line vectors must be compatible with other sub-vectors, we require the start and end number-line vectors in a range (maximally distant) to have a $HSim = 0.5$ (orthogonal vectors have $HD = HDS = 0.5$) and hence only half of the vector bit count (vector dimension) can be used when generating the bit-flip interval required between vectors. Figure 5.1 shows the hamming similarity between points on a 1D number-line.

Listing 5.1: Generate number-line vectors.

```
def linear_sequence_gen(max_number, start_vec, orth_vec):
    # max_number : terminal number
    # start_vec : base starting vector
    # orth_vec : a unique role vector used to move the numberline set into its own space
    # return list of vectors having equal hamming distances between each vector
    if max_number > vec_len:
        assert False, "Ranges > number of bits should be normalized"
    vec_len = len(start_vec) // 2 # Orthogonal vecs are (vec_dim / 2) bits apart
    number_line = [start_vec]
    change_map = np.zeros(len(number_line[0]), dtype=int) # Track flipped bits
    bits_to_flip = vec_len // max_number # bit interval between steps
    for i in range(1, max_number+1):
        # Randomly flip bits in the previous vector to create the next vector
        z, change_map = gen_next_vec(number_line[i−1], change_map, bits_to_flip)
        number_line.append(XOR(z, orth_vec))

    return number_line # Ordered list of vectors
```

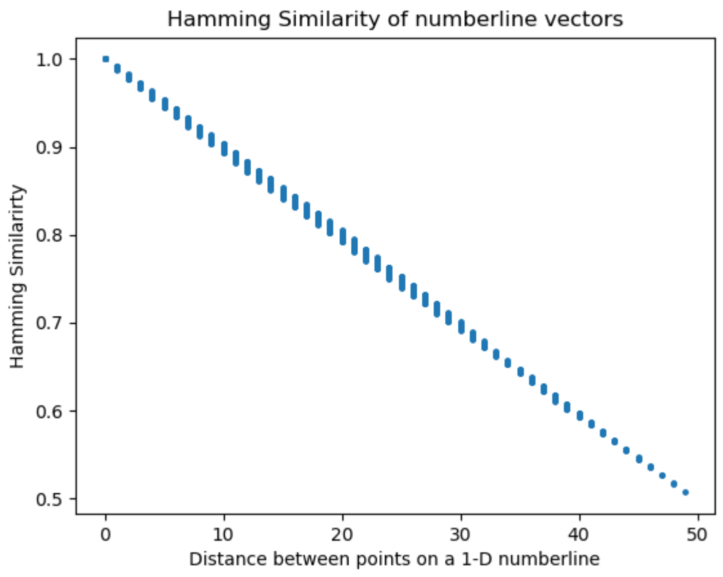


Figure 5.1: Hamming Similarity of vectors in a number-line set.

Because *XOR* binding preserves distance, we can create vectors that represent points on a 2D plane by binding the x, y coordinates of a point together. The hamming similarity between such *'2D point vectors'* is approximately linear except for the extremes of distance as shown in Fig. 5.2 (the non-linear portion can be excluded by restricting acceptable values to a subset of a larger range).

$$Point(x, y) => XOR(Xnumberline[x], Ynumberline[y]) \tag{5.6}$$

The 1D number-line and 2D Point vectors are a very simple way of *'semantically'* representing numerical values. The overlap / mistakes made (when the hamming similarity result suggests the wrong match as the closest) are small, in the sense of *'semantic matching'* by value. That is, we are able to make requests such as *"find something that*

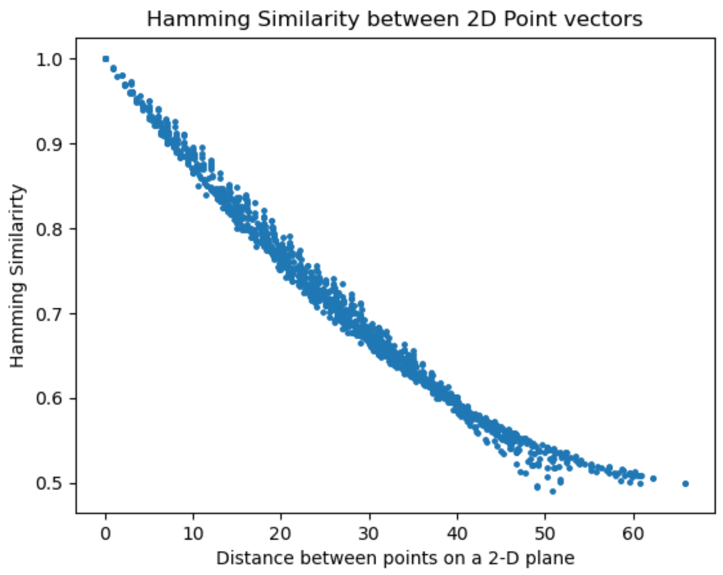


Figure 5.2: Hamming Similarity between points on a 2D plane.

is near this value(point)". In order to enable proximity based matching, the required position vector, let us call it $vsa\_requested\_posn$ (encoded using Eq. 5.6), is added to the workflow vector Z_x as metadata before it is transmitted as a request vector. This metadata parameter can be recovered from the workflow vector by each node on receipt of the workflow request. On receipt of a workflow request each node can compare its static node description vector, $static\_match\_vec$, to the static request vector, weighted for position as follows:

$$proximity\_vec = XOR(vsa\_requested\_posn, vsa\_node\_pos) \tag{5.7}$$

$$static\_match\_vec = XOR(workflow\_vec, node\_desc\_vec) \tag{5.8}$$

$$similarity = HSim(proximity\_vec, static\_match\_vec) \tag{5.9}$$

When vsa_node_pos and $vsa\_requested\_posn$ are similar to each other, the result of Eq. 5.7 will be mainly zeros. Similarly when $workflow\_vec$ and $node\_desc\_vec$ are similar, the result will be mainly zeros. Taking the hamming similarity between the two, Eq. 5.9 modifies the $static\_match\_vec$, weighted by proximity.

This technique can be used to perform weighted matching on any scalar attribute. For example, battery life can be represented as a 1D number-line, and the same operation, Eq 5.9, can be used to obtain a vector weight mask for battery life. Multiple weighted matches are carried out by applying each in a chained XOR fashion. To match on battery life as well as position we simply perform the following:

$$battery\_life = XOR(vsa\_required\_batt, vsa\_node\_batt) \tag{5.10}$$

$$proximity\_vec = XOR(vsa\_requested\_posn, vsa\_node\_pos) \tag{5.11}$$

$$comb\_weight\_vec = XOR(battery\_life, proximity\_vec) \tag{5.12}$$

$$static\_match\_vec = XOR(workflow\_vec, node\_desc\_vec) \tag{5.13}$$

$$similarity = HSim(comb\_weight\_vec, static\_match\_vec) \tag{5.14}$$

There is no restriction on the number of scalar QoS attributes that can be combined in this way, not withstanding that as more QoS attributes are chained together, there is less chance, as to be expected, of a node matching all requirements.

5.3 Using existing semantic vector representations (R1)

Data analysis and machine learning methods use many feature extraction techniques that ultimately output real number vectors to describe the input object [105]. Data including images, sound and text can be vectorized in various ways. These vectors might then be used as input to another analysis or machine learning stage. For example, the flattened output vector of a CNN's convolution layers (a vector representation of the input image) is fed to its fully connected classifier stage. In NLP, words and sentences are often vectorized for further manipulation and analysis. Word2Vec [106][107] takes a text corpus as input and generates real number output vectors for each word in the corpus such that the words having similar meaning have vectors that are closer together in the output vector space.

It is noted that real number vectors can be converted to BSCs using the method of randomized binary projection [108]. The conversion method also transfers the spatial relationships between vectors within the real number vector space to the binary output vector space. This means that ready made, leading edge, semantic word representation such as the pre-trained Google News corpus [106] can be leveraged for the purpose of

BSC service description. Via the method of random binary projection, it is also feasible to create multi-modal service description vectors. For example, an alarm detection service could combine a textual word2vec semantic vector description with vectorized sound samples of the type of noise that the service should detect.

Listing 5.2 provides an algorithm for converting any dimension real number vectors to BSCs by taking the scalar product between a random binary matrix and the real number input vectors. Table 5.1 gives a comparison of the cosine distance wordrank search on the word *'weather'* from the GoogleNews-vectors-negative300.bin word2vec database when loading the first 1M words compared to the same database converted to BSCs. The real number input vectors have size $rdim = 300$ and the BSC output vectors had size $bdim = 10000$.

Listing 5.2: Real2Binary.

```
class Real2Binary(object):
    def __init__(self, rdim, bdim, seed):
        """
        Note when converting a 'bank' / database of realnumber vectors the same seed MUST be used
        in order to ensure that the semantic vector space distances are maintatined.
        Obviously a single run will maintain this since we generate the mapper on initialisation.

        :param rdim: Dimension of the real number vec being converted
        :param bdim: Dimension of the equivalent binary vector we want to create
        :param seed: for repeatability if needed during research and debug etc
        """
        if seed:
            np.random.seed(seed)
        self.mapper = np.random.randint(0, 2, size=(bdim, rdim), dtype='uint8')

    def to_bin(self, v):
        """
        To create the binary vector take the scalar product between the mapper matrix
        and the real number vector. The random bit patterns in self.mapper * v produces
        a (bdim * rdim) real number matrix.
```

```
we then sum along axis=1 which gives us a 'bdim' real number vector.
This is then thresholded to produce a binary bit pattern that maintains distances in
the vector space. The binary vector produced has an, approximately, equal number
of 1's and 0's thus maintaining the i.i.d random distribution of bits within the vector.

Example,
            2d real number vec R = [0.3, -0.7]
            5D binary mapper B = [[1, 0],
                                  [1, 1],
                                  [0, 0],
                                  [1, 0],
                                  [1, 1]]

                           R * B = [[0.3, 0],
                                  [0.3, -0.7],
                                  [0.0, 0.0],
                                  [0.3, 0.0],
                                  [0.3, -0.71]

    Sum R * B along axis1 => RB = [0.3, -0.4, 0.0, 0.3, -0.4]

    We then perform thresholding and normalisation on 'RB' to convert this to a
    binary presentation ZZ,
                               ZZ = [1, 0, 1, 1, 0]

:param v: real number vector to convert.
:return: Binary vector representation of v having an i.i.d, approx equal number of 1's and 0's.
"""
Exp_V = 0.5 * np.sum(v)
Var_V = math.sqrt(0.25 * np.sum(v * v))
ZZ = (np.sum(self.mapper * v, axis=1) - Exp_V) / Var_V # Sum and threshold.
# Normalize this to binary
ZZ[ZZ >= 0.0] = 1
ZZ[ZZ < 0.0] = 0
```

```
        return ZZ.astype('uint8')
```

We can see from Table 5.1 that the conversion to BSCs effectively maintains the semantic relationships between vectors. For example, a BSC query for the idea *'weather service'* will match not only to the word *weather* but also words such as *inclement*, *fog* and *thunderstorms*. Because the quality of similarity weights the comparisons, we will obtain better matches on service objects that directly specify the word *weather* than those that only specify synonyms.

rank	name	Cosine Real no.	name	HSim BSCs
0	weather	1.0000	weather	1.0000
1	inclement	0.8251	inclement	0.8101
2	fog	0.7592	fog	0.7714
3	daylight	0.7584	daylight	0.7679
4	thunderstorms	0.7065	thunderstorms	0.7526
5	humidity	0.7015	humidity	0.7442
6	snowstorms	0.7005	snowstorms	0.7434
7	forecasters	0.6957	breezes	0.7396
8	nighttime	0.6932	overcast	0.7393
9	overcast	0.6863	nighttime	0.7373

Table 5.1: Top 10 nearest words to *'weather'* from 1M word GoogleNews word2vec database and corresponding Hamming Similarity for the same 1M words converted to a BSC database.

5.4 Encoding JSON / XML service descriptions

In Simpkin et al. [3], the Service Vector (SV) encoding scheme described in Section 5.1 was extended to show how role-filler pair binding and bundling can be used to create description vectors directly from a JSON service description file. (Section 9.1.5 describes the majority of this paper; however, the SV description section is described here.) The method applies equally well to the encoding of any format of key-value plain text data files. Unique role vectors are built from the sequence of nested keys. These are then combined with their associated JSON value fields to create a bag of role-filler pairs that is majority summed into a SV that describes the service.

Listing 5.3 is an example of a JSON service description for one of the Node-RED object detectors in the Node-RED Traffic Congestion use case (Section 9.1.5).

Listing 5.3: Service Vector Description

```
{"service":
  {"service_name":"object_detector_1",
    "service_inputs":[
       {"input_name":"image",
        "input_data_type":"char64jpg",
        "input_related_concepts":[{
             "concept_name":"location"}],
         "required":true}],
    "service_outputs":[
       {"output_name":"object_list",
        "output_data_type":"list_string",
        "output_related_concepts":[
           {"concept_name":"car"},
           {"concept_name":"person"},
           {"concept_name":"bus"}]}],
    "service_average_response_time_ms":5000}}
```

The key-names within the JSON must be converted to unique role vectors, and the JSON values-fields are converted to semantically comparable vector values. When using role vectors generated from JSON key-names to categorise the feature values of a service vector concept, one important question is how to guarantee that the role vectors created are unique and have the same value across distributed service implementations? This is a particularly relevant question for the Node-RED integration test-case since Node-RED is open source and functional nodes/services can be created arbitrarily by an unrelated set of developers. One approach is to simply assign, known, random hyper-dimensional vectors to each role/key-name. However, this does not allow for unrelated developers to invent new key-names and would require some sort of central database lookup so that distributed services can agree on the vector value of a role/key-name, otherwise they would not be able to perform semantic matching.

Simpkin et al. [3] describes an alternate vector encoding method that ensures roles are always unique based on their case insensitive spelling. The encoding algorithm used for the key-names is *chained XOR* of a shared vector alphabet. *Cyclic-shift* per character position is used to ensure unique encodings for words such as 'AA' and 'AAA', which would otherwise collapse into similar values. The collapse is because, $XOR(A,\ A) = \hat{0}$ and $XOR(XOR(A,\ A),\ A) = A$. The algorithm to convert a field name to a vector is shown in Listing 5.4.

Listing 5.4: Field Name to Vector.

```
def field_name_to_vec(name, vec_alphabet):
    n = name.lower()
    v = vec_alphabet[n[0]]
    shift = 0
    for c in n[1:]:
        shift += 1
        v = XOR(v, ROLL(vec_alphabet[c], shift))
    return v
```

The $json\_to\_vecs()$ method shown in Listing 5.5 recursively encodes a nested JSON by chaining together the key and sub-key vectors, created by $field\_name\_to\_vec()$, together with the associated JSON value, created using $CreateValueVector()$.

Listing 5.5: Chaining Field Names.

```
def json_to_vecs(json_input):
    if isinstance(json_input, dict):
        dd = []
        for k, v in json_input.iteritems():
            rv = json_to_vecs(v) # Recurse
            if isinstance(rv, list):
                dd.extend([("{} ∗ {}".format(k, i[0]),
                    # Chain XOR key−names with
                    # sub role−filler found in i[1]
                    XOR(field_name_to_vec(k, symbol_dict), i[1]))
                    for i in rv])
            else:
                dd.append(("{} ∗ {}".format(k, rv[0]), XOR(
                   field_name_to_vec(k, symbol_dict), rv[1])))
        return dd
    elif isinstance(json_input, list):
        dd = []
        for item in json_input:
            rv = json_to_vecs(item) # Recurse
            if isinstance(rv, list):
                dd.extend([json_to_vecs(i) for i in rv]) # Recurse
            else:
                dd.append(rv)
        return dd
    else:
        if isinstance(json_input, tuple):
            return json_input
        else:
            return json_input,
                CreateValueVector(str(json_input)).myvec
```

The $CreateValueVector()$ method creates semantically comparable vectors. It can be customized to create semantic vectors from text strings using a vector alphabet and suitable encoding scheme, for example eq. (6.1), or by employing existing semantic word representations as described in Section 5.3. JSON/XML number values can be encoded using the *role* and *range* methods described above, or the *number-line* method described in Section 5.2. The algorithm produces a 'bag' (python list) of role-filler vectors that are then further combined into a single, semantically comparable vector using simple $majority\_vote$ addition. The output of $json\_to\_vecs()$ for JSON Listing 5.3 is shown in schematic form in Listing 4.

Listing 5.6: Output from $json\_to\_vecs()$.

```
service ∗ service_name ∗ object_detector_1
service ∗ service_average_response_time_ms ∗ 5000
service ∗ service_inputs ∗ input_data_type ∗ char64jpg
service ∗ service_inputs ∗ input_related_concepts ∗ concept_name ∗ location
service ∗ service_inputs ∗ required ∗ True
service ∗ service_inputs ∗ input_name ∗ image
service ∗ service_outputs ∗ output_data_type ∗ list_string
service ∗ service_outputs ∗ output_name ∗ object_list
service ∗ service_outputs ∗ output_related_concepts ∗ concept_name ∗ car
service ∗ service_outputs ∗ output_related_concepts ∗ concept_name ∗ person
service ∗ service_outputs ∗ output_related_concepts ∗ concept_name ∗ bus
```

Note, in the listing $'*'$ indicates XOR binding.

Each line in Listing 5.6 represents a compound vector entry in the returned list. The right most vector is the value vector, all vectors to the left of this are unique role vectors. During recursion of $json\_to\_vecs()$, vectors are returned in right to left order, so that the resultant vector is built by XOR chaining from right to left as shown:

$$subfeature_v = service\ *\ (service\_name\ *\ (object\_detector\_1))$$

In the above example, $object\_detector\_1$ is the value vector and $service$ and $service\_name$

are both role vectors. If Z_v is the result of the final $majority\_vote$ superposition, then to extract a noisy copy of the $object\_detector\_1$ value, we would perform

$$object\_detector\_1 \approx XOR(service\_name_r,\ XOR(Z_v,\ service_r))$$

Note, as mentioned above, that the output of $json\_to\_vecs()$ is combined as a simple $majority\_vote$ bag of vectors, this helps make the vectorisation of JSON service descriptions immune to ordering issues but does limit the number of service line entries to approximately 100; the maximum capacity of a single 10kbit binary vector [46].

In Node-RED such vector encodings are representative of the required function. The encoded JSON may be a specific known function that has been previously used or a generic JSON representing the type of functional service needed to fulfil a workflow service step.

5.5 Limitations

The encoding methods described in Sections 5.1 and 5.4 for the creation of Service Vector (SV) descriptions are effectively vector representations of unordered collections of key-value attribute pairs. As such, these type of SV encodings can offer only shallow semantic, i.e. syntactic matching, capability. A further work objective is to consider how deeper semantic representations can be captured. This is a particularly challenging requirement because very often true semantic similarity depends on context. For example, consider the challenge of finding a camera sensor near a particular location, say the north end of Oxford Street. We can specify a lat-long as a key-value entry in a JSON camera resource description file and use the proximity matching technique described in Section 5.2.1 to encode the location. If the objective of the workflow is to search for a vehicle of interest near the location specified, then any camera sensors returned near the lat-long specified, including cameras in adjacent streets, might be considered a good match. However, if the workflow objective is to obtain video analysis of an incident specifically happening at the north end of Oxford street, then only cameras with a view of Oxford Street would be acceptable. Thus, in the second scenario, a camera located half a mile farther down Oxford street might be an acceptable match but very close cameras in adjacent streets would not.

One possible avenue for endowing services with deeper semantic meaning is described in Section 6.2.6; that is, when service objects participate in workflows, the contexts in which they have previously been invoked can be used to learn similarities between self and other service objects. In this way when a service ‘sees’ a SV request for a service that it does not exactly match, it can use the knowledge that it has been invoked in the same context (i.e. between the previous and subsequent workflow steps) to consider itself a match.

Another might be to explore how query by example could be encoded in SV descriptions? That is, is it possible in some way to capture the transfer function of a microservice, considered as a black box, by encoding examples of the typical input it accepts and corresponding output it produces?

5.6 Summary

This chapter has described creating BSC Service Vector (SV) descriptions (R1) to represent operational microservices as a binary vector embedding of multiple sub-feature vectors. Rich multi-model descriptions of service objects and sensors can be created by encoding the object’s sub-feature vectors from a combination of simple VSA *binding* and *bundling* operations as well as from existing semantic vector word databases such as word2vec and non-text based vector descriptions. Matching can also be performed on approximate values using the range and ‘number-line’ methods. Binding and bundling operations are mathematically well defined; therefore, similarity comparisons are deterministic and explainable.

Using hamming similarity to compare service descriptions represented as BSC vectors is a significant advantage over the complex processing necessary when attempting to compare service descriptions expressed in conventional terms such as JSON/XML or ontological languages because hamming similarity is a one-shot operation. It returns a graded match value and is also agnostic to unknown terms and term ordering (due to the encoding scheme presented) and so does not need to be changed or updated to cater for new description keys and so forth.

The individual service steps required to perform a particular workflow are also encoded as described here, that is, a BSC compound/concept vector representing the sub-features that a particular workflow step is required to meet. When the workflow is executed, these workflow service steps are matched to the service descriptions created by each listening service on the network. The next chapter explains how workflows can be encoded as ordered sequences of BSC service request vectors.

Chapter 6

Cognitive Workflows using a Vector Symbolic Architecture

This chapter describes a new VSA encoding scheme (R2)that can be used to scale linear and Directed Acyclic Graph (DAG) workflows to a practically unlimited number of workflow steps. The ability to scale is a major advantage since it is not unusual for workflows to require many hundreds of individual steps in today's microservice approach to workflows. In addition, the new encoding encapsulates all the information needed to control workflow discovery and execution without a central controller and without the need to specify the IP-addresses of the required services and maintains semantic matching capabilities at each level in the hierarchy (workflow and sub-workflow matching).

6.1 Recursive embedding - Chunking

Distributed workflows, particularly in IoBT scenarios, are likely to consist of many functional microservice and sensor objects. For this reason, if we want to encode workflows using a VSA, it is necessary to overcome the sub-vector 'storage' capacity limits that are a consequence of the lossy *majority_sum* BSC bundling operation described in Sections 4.1, 4.2, and 4.3. A recursive method for embedding vectors within vectors is therefore required. Such a method must also maintain the ability to perform matching at each level of recursion. In general all recursive embedding, or *chunking*, schemes employ the same approach. If an object has too many sub-feature vectors to be bundled into a single compound vector, then the sub-feature vector list is split into smaller groups that are bundled into separate vectors. These vectors are then used as the basis for further chunking operations, thus recursively producing a hierarchical tree structure as shown in Figure 6.1.

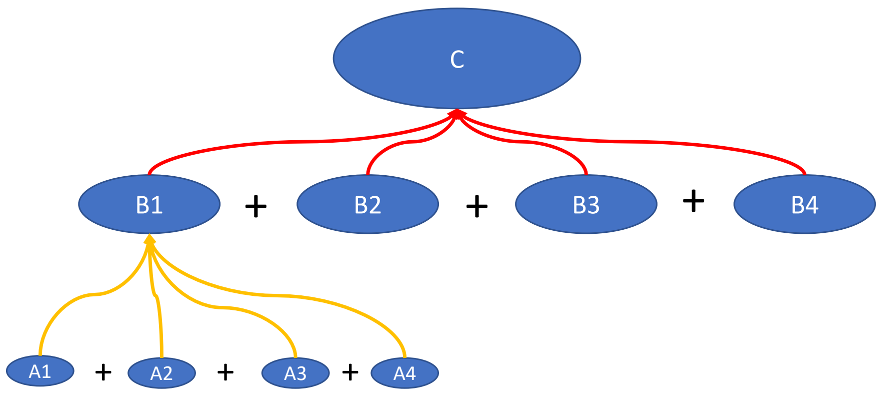


Figure 6.1: Vector Chunk Tree, chunking proceeds from the bottom up

In the diagram, the maximum sequence length is set at four. Chunking proceeds from the bottom up, bundling the A-node vectors into groups of four. This produces the B-node 'parent' or non-terminal vectors which are summed into the single parent vector, C. In order to be able to traverse the hierarchy, each 'parent' vector must be somehow made available for use as a *clean-up memory* for the level below. For example, the bundled vector C contains noisy copies of $B1, B2, B3, B4$. To traverse from C to $A1$ a 'clean' copy of $B1$ is needed since the representation of $A1$ contained directly in C maybe too noisy to correctly decode it (or, when thinking of services, to activate it). In a centralized system, the intermediate parent nodes are simply stored. For a VSA workflow sequence, the idea of instantiating the intermediate nodes as *clean-up services* is introduce that act as a proxy to the terminal node worker services. When activated, a clean-up service unbinds its clean vector, for example, $B1$, which will, when transmitted to the network, activate $A1$ and so forth, as described in Section 6.2.2.

6.2 Hierarchical VSA Workflows (R2)

Various methods of recursive vector embedding have been described [36, 41, 46, 48]. However, as demonstrated in Section 4.5, such methods suffer from limitations when employed for multilevel vector embeddings: some lose their semantic matching ability even if only a single term differs, while others cannot maintain separation of sub-features for higher level compound vectors when lower level chunks contain the same vectors [48, page 148] [36, pages 61, 72, 74–para2] [46, Encoding Sequences, page 14]. One of the linchpins of this thesis is now presented; a novel hierarchical VSA binding and bundling scheme that enables fully recursive embeddings of vectors (R2).

The new encoding scheme, shown in eq. (6.1), employs both XOR and cyclic-shift binding to enable recursive bundling that supports a practically unlimited hierarchical vector embedding scheme capable of encoding any number of sub-feature vectors even when there are repetitions and similarities between sub-features:

$$Z_x = \sum_{i=1}^{cx} \left(Z_i^i \cdot \prod_{j=0}^{i-1} p_j^0 \right) + StopVec^{cx} \cdot \prod_{i=0}^{cx} p_i^0 \tag{6.1}$$

Omitting $StopVec$ for readability, this expands to,

$$Z_x = p_0^0 \cdot Z_1^1 + p_0^0 \cdot p_1^0 \cdot Z_2^2 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot Z_3^3 + \dots \tag{6.2}$$

Where

- "$\cdot$" is defined as the XOR operator.
- "$+$" is defined as the Bitwise_Majority_Vote/Add operator.
- The exponentiation operator is redefined to mean cyclic-shift—i.e., positive exponents mean C_{shift_right}, negative exponents mean C_{shift_left}. Note that cyclic shift is key to the recursive binding scheme since it distributes over bitwise majority addition and XOR. Hence, it automatically promotes its contents into a

new part of the hyper-dimensional space, thus keeping levels in the chunk hierarchy separate.

- Z_x is the next highest semantic *chunk* item containing a *superposition* of x sub-feature vectors. Z_x chunks can be combined using eq. (6.1) into higher level chunks. For example, Z_x might be the superposition of $B1 = \{A1, A2, A3, \ldots\}$ or $C = \{B1, B2, B3, \ldots\}$.
- $\{Z_1, Z_2, Z_3, \ldots Z_n\}$ are the sub-feature vectors being combined for the individual nodes of Figure 6.1. Each Z_n itself can be a compound vector representing a sub-workflow or a complex vector description for an individual service step, built using the methods described in Section 5.1.
- $p_0, p_1, p_2, \ldots$ are a set of known *atomic* role vectors used to define the current position or step in the workflow.
- cx is the chunk size of vector Z_x, i.e., the number of workflow service request vectors being combined.
- $StopVec$ is a role vector owned by each Z_x that enables it to detected when all of the steps in its (sub)workflow have been executed.

6.2.1 Creating a workflow

Equation (6.1) is used recursively to build a workflow request, conceptually creating a hierarchical chunk tree as shown in Figure 6.1. The resulting output is a set of VSA vectors representing the non-terminal *clean-up* nodes, $\{C, B1, B2, B3, \ldots\}$, each of which is a single VSA vector Z_x that is itself a compound vector representing the ordered sequence of its own children.

$$
\begin{aligned}
C &= p_0^0 \cdot B_1^1 + p_0^0 \cdot p_1^0 \cdot B_2^2 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot B_3^3 + ...B_4^4 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot p_3^0 \cdot p_4^0 \cdot C_{StopVec}^5 \\
B1 &= p_0^0 \cdot A_1^1 + p_0^0 \cdot p_1^0 \cdot A_2^2 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot A_3^3 + ...A_4^4 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot p_3^0 \cdot p_4^0 \cdot B1_{StopVec}^5 \\
B2 &= p_0^0 \cdot A_5^1 + p_0^0 \cdot p_1^0 \cdot A_6^2 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot A_7^3 + ...A_8^4 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot p_3^0 \cdot p_4^0 \cdot B2_{StopVec}^5 \\
B3 &= ...\ etc
\end{aligned}
$$

The reason multiple p vectors are XOR chained together to define a single position within the workflow is due to the distributive property of XOR which operates on every term for each unbinding (see, eq. (6.4) and eq. (6.6)). Thus, the use of the chained p vectors when constructing the workflow vector allows for easier iterative unbinding at execution time using eq. (6.5), discussed below. The generalized version of the concept vectors, $\{C, B1, B2, \ldots\}$, is shown in eq. (6.2). Note that in this form, every sub-step Z_n is permuted by at least one p vector that effectively hides each Z_n (the p vector permutation ensures that each combined role-filler pair is orthogonal to the 'self-description' vectors built by each VSA service listening for work on the network). The workflow is discovered and orchestrated on the distributed services by, essentially, repeatedly *unbinding* the workflow vector, using eq. (6.3) or eq. (6.5), before re-transmitting it from peer-to-peer.

6.2.2 Ordered Unbinding of High-level Concept Vectors

Microservices such as B1, B2, B3, B4 are termed *clean-up* services. They are used to maintain the decodability of large workflows that have been segmented by chunking. They act as an activation path through to the terminal node services that ultimately perform the desired operational workflow steps. On becoming activated, by matching to a noisy version of its SV, such a *clean-up* microservice broadcasts a '*clean*' copy of its SV which will, in turn, cause its child sub-tree to execute. For large workflows control may pass down through multiple levels of clean-up services before work is carried out at the terminal nodes of a chunk tree. Referring to Figure 6.1, control first passes down the chunk tree, i.e., from $C \rightarrow B1 \rightarrow A1$ using eq. (6.3), before traversing horizontally, $A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow B1\_StopVec$) via eq. (6.5). At this point, $B1$ *sees* its $StopVec$ and employs eq. (6.5) to activate $B2$, which then activates its sub-workflow via eq. (6.3) and so forth.

Starting a (sub)workflow:

$$Z_1' = (p_0^0.[T + M_{data} + Z_x])^{-1} \tag{6.3}$$

$$Z_1' = p_0^{-1}.(T^{-1} + M_{data}^{-1}) + \boxed{Z_1^0} + p_1^{-1}.Z_2^1 + p_1^{-1}.p_2^{-1}.Z_3^2 + ... \tag{6.4}$$

Traversing horizontally:

$$Z_{n+1}' = (\mathbf{p_n^{-n}} \,.\, \mathbf{Z_n'})^{-1} \tag{6.5}$$

$$Z_2' = (p_1^{-1}.Z_1')^{-1} = p_1^{-1}.p_0^{-2}.(T^{-2} + M_{data}^{-2}) + p_1^{-1}.Z_1^{-1} + \boxed{Z_2^0} + p_2^{-2}.Z_3^1 + \tag{6.6}$$

When starting a (sub)workflow using eq. (6.3), notice that Z_1 has been exposed, as shown in eq. (6.4). That is, Z_1' is effectively a noisy copy of the currently required workflow step, Z_1, while at the same time it is also a unique permutation of the full (sub)workflow request. Thus, listening services can only match to Z_1' if they are semantically similar to Z_1. Note that the act of matching gives a service no other information; for example, it cannot deduce by matching alone whether the match occurred at step 1 or step 30 of the workflow. Hence, the introduction of the T vector in eq. (6.3) which is used to enable calculation of a node's position within the workflow.

T vector and Metadata

The T vector is a known *atomic* role vector. It is added, using *late-bundling*, to a high level node's clean (sub)workflow vector, Z_x, before the node uses eq. (6.5) to expose its first workflow step for transmission to the network. We note that eq. (6.3) is just a special version of eq. (6.5). Notice in eq. (6.4) and eq. (6.6) how the T vector becomes permuted in a predictable way. Once the currently active service has completed its own workflow step, it uses the current permutation of the T vector to calculate its position 'n' within the received request vector. It can then activate the next workflow step in the request by repeating the *unbind* operation on the request vector, generalized in eq. (6.5). Thus, the workflow proceeds in a completely decentralized manner whereby each node is activated when its preceding node, or parent, *unbinds* the currently active chunk vector, creating the next request vector, which it then multicasts to the network for matching and processing.

Workflow metadata, M_{data}, can be added by a clean-up service that is starting its workflow vector. It is added, using *late-bundling*, at the same level as the T vector. Because the current permutation of the T vector is easy to calculate (simply permute T forward or Z'_n backwards using eq. (6.3) or eq. (6.5) and check for a match), the workflow metadata is easy to retrieve.

Alternative mechanisms for determining the position of the service might be to permute forward or backwards checking for a match with '*self*', i.e the matching service's SV. However, this would significantly increase the work that each service has to perform. In addition, services that are not a match to any step in the workflow would be unable to discover the metadata vector. The ability to inspect the metadata of any received SV, via an agreed upon '*tag*' vector, is useful to the Service Vector Architecture (SVA) since it can simplify node operations and be used to allow listeners to remain cognizant of the networks distributed workflow activity as a whole.

6.2.3 Chunk Stop Vector

Continuing with the workflow example, when all the workers of B1 have completed, B1 must activate B2 in order to continue the workflow. Since the B1 service knows the value of the B1 vector and understands the workflow progression in use, it can easily monitor the sub workflow execution. It simply tracks the modification and rebroadcasting of B1 as the child services pass control from A1 through A4; however, the last broadcast it will see is when A3 activates A4. Thus, the B1 service knows that A4 has started but it needs a mechanism to determine when A4 completes. This is achieved by appending a STOP vector to the B1 list of vectors being added during chunking. Returning to the example, when A4 completes, it performs an unbind (as does every node on completion of their task), which results in the multicast of the B1 STOP vector. On recognising its own STOP vector, the B1 service performs an unbind on the C vector (since it recognises that its task is complete) and rebroadcasts this to activate the B2 clean-up service and so on.

The simplest STOP vector implementation is to have the parent clean-up service add a *stop* role vector to the end of its workflow sequence. The stop vector added must be unique to each clean-up service in order to avoid several pitfalls:

1. To avoid parallel executing workflows detecting false stop signals.
    - In practice, these types of clashes are better resolved using a vector *job_id* metadata parameter.
2. If a single role vector, e.g., $StopVecConst$, common to all services is used as the stop vector, this will cause erroneous hamming similarity comparisons, especially for short workflows. For example, all one-step workflows (e.g., $S1 = p0.Z_1^1 + p0.p1.StopVecConst$ and $S2 = p0.W_1^1 + p0.p1.StopVecConst$) will have $HSim(S1_1, S_2) \approx 0.625$ instead of 0.5!

However, using unique stop vectors would inhibit other listening services from detecting stop vector signals including the worker service that is activated by the final step of the workflow (the stop vector is owned and added by the parent requester/clean-up service). For this reason, the stop vector is encoded as follows.

$$StopVec = role\_stopvec \cdot Z_{last}^1 \tag{6.7}$$

where $role\_stopvec$ is a known role vector and Z_{last}^1 is the last service step request vector. This has two benefits.

1. The last service in a sub-workflow, on unbinding the next vector, can detect that it is about to send the stop vector. This is needed in the implementation because the normal action of a node on unbinding and sending the next vector is to enter local arbitration mode and wait for replies. However, if the node is in last position it should not do this.

2. The use of the last workflow request vector Z^1_{last}, enables better matching for workflows that end on the same workflow request step in contrast to the idea of using a single value *StopVec* as discussed in point (2) of the previous list above. The use of a permuted copy of the final service step improves matching without causing erroneous matching.

The $StopVec$ mechanism can be further leveraged for specific applications to signal different actions when the end of a workflow is detected. For example, using different values for the $role\_stopvec$ can trigger different behaviours such as looping.

6.2.4 Hierarchical VSA Encoding of DAG Workflows

By way of example, this section describes how the vector representation can be extended to more complex workflows such as those created by the Pegasus workflow generator. Figure 6.2 shows a typical Pegasus workflow (the Montage Workflow) having multiple connections between nodes with branching and merging of connections.

In order to represent such DAGs, we modify our linear scheme by employing a three-phase process comprising the following:

1. A recruitment phase, where the required services are discovered, selected and uniquely rename themselves;

2. A connection phase, where the selected services connect themselves together using the newly generated names; and

3. An *atomic start* command that indicates to the connected services that the workflow is fully composed and can be started.

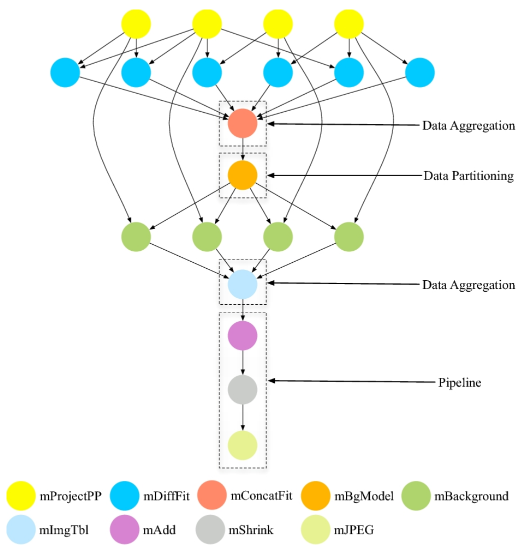


Figure 6.2: Montage Workflow

```
<job name="mProjectPP" ... id="ID00000"> ... </job>
<job name="mProjectPP" ... id="ID00001"> ... </job>
                        :
<job name="mDiffFit" ... id="ID00004"> ...</job>
<job name="mDiffFit" ... id="ID00005"> ...</job>
                        :
<child ref="ID00004">
  <parent ref="ID00000"/>
  <parent ref="ID00001"/>
</child>
<child ref="ID00005">
  <parent ref="ID00000"/>
  <parent ref="ID00001"/>
</child>
```

Listing 6.1: DAX snippet

The workflow shown in Figure 6.2 can be represented as a symbolic vector as follows:

$$WP \ = p0^0 \cdot (Recruit_{Nodes})^1 + p0^0 \cdot p1^0 \cdot (Connect_{Nodes})^2 + \ p0^0 \cdot p1^0 \cdot p2^0 \cdot Start^3$$

where:

$$\begin{aligned} Recruit_{Nodes} = {} & p0^0 \cdot Z_1^1 + p0^0 \cdot p1^0 \cdot Z_1^2 + \ldots 0^0 \cdot p1^0 \cdot p2^0 \cdot p3^0 \cdot Z_1^4 \\ & + p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p4^0 \cdot Z_2^5 \ldots + \ p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p9^0 \cdot Z_2^{10} \\ & + p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p10^0 \cdot Z_3^{11} \ + \ p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p11^0 \cdot Z_4^{12} \\ & + p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p12^0 \cdot Z_5^{13} \ + \ p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p15^0 \cdot Z_5^{16} \\ & + p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p16^0 \cdot Z_6^{17} \ + \ p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p17^0 \cdot Z_7^{18} \\ & + p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p18^0 \cdot Z_8^{19} \ + \ p0^0 \cdot p1^0 \cdot p2^0 \ldots \cdot p19^0 \cdot Z_9^{20} \end{aligned}$$

$$Connect_{Nodes} = \left(p0^0 \cdot \mathbb{P}_1^1 + p0^0 \cdot p_1^0 . \mathbb{C}_1^2\right) + \left(p0^0 \cdot p1^0 \cdot p2^0 \cdot \mathbb{P}_2^3 + p0^0 \cdot p1^0 \cdot p2^0 \cdot p3^0 \cdot \mathbb{C}_2^4\right) \cdots$$

Each $\boldsymbol{Z_n}$ in $\boldsymbol{Recruit_{Nodes}}$ is the compound vector representation of each service. In our implementation, the vectors are constructed automatically from an XML workflow definition file, the Pegasus DAX file, see [109] for more details. An example DAX snippet is given in Listing 6.1. The $\boldsymbol{Recruit_{Nodes}}$ vector is built from the <job> entries found in the DAX—we refer the reader to Figure 6.2 and Listing 6.1, where there are four mProjectPPs$(\boldsymbol{Z_1 s})$, six mDiffFitt$(\boldsymbol{Z_2 s})$ and so on.

The $\boldsymbol{Connect_{Nodes}}$ vector, built from the <child> entry section of the DAX, defines the producer/consumer relationship between nodes. A node can act as both a parent (producer) and child (consumer) within the workflow (see Figure 6.2). Using Listing 6.1 as an example, the parent, $\mathbb{P}$, and child, $\mathbb{C}$, ends of each edge are constructed as follows:

$$\begin{aligned}
\mathbb{P}_1 &= \boldsymbol{Z_1^0} \cdot \left(\boldsymbol{NodeID_r^0} \cdot \boldsymbol{Parent_r}\right) \\
\mathbb{C}_1 &= \boldsymbol{Z_2^0} \cdot \left(\boldsymbol{NodeID_r^4} \cdot \boldsymbol{Child_r}\right) \\
\mathbb{P}_2 &= \boldsymbol{Z_1^0} \cdot \left(\boldsymbol{NodeID_r^1} \cdot \boldsymbol{Parent_r}\right) \\
\mathbb{C}_2 &= \boldsymbol{Z_2^0} \cdot \left(\boldsymbol{NodeID_r^4} \cdot \boldsymbol{Child_r}\right)
\end{aligned}$$

where

- $\boldsymbol{NodeID_r^n}$ is an *atomic* role vector used to encode a node's integer *id* as defined in the DAX. For this purpose we encode an integer $\boldsymbol{i}$ as a single *atomic* role vector cyclic shifted by $\boldsymbol{i}$, for example, $\boldsymbol{NodeID_r^4} = (\boldsymbol{int})\ 4$, see section 5.2.
- $\boldsymbol{Parent_r}$ and $\boldsymbol{Child_r}$ are fixed *atomic* role vectors used to bind the resultant vector into the parent or child category.
- $\boldsymbol{Z_1}$ represents mProjectPP and $\boldsymbol{Z_2}$ represents mDiffFitt, as described above.

By binding these three elements together, we construct a unique encoding for the parent and child ends of every edge in the DAG. Using eq. (6.1) we then represent the edges as an ordered list of parent→child ends, (see $\boldsymbol{Connect_{Nodes}}$ above).

Execution of the workflow

The resulting workflow $\boldsymbol{WP}$ is a superposition representing the linear sequence of steps needed to: *(a)* discover the required services, *(b)* connect the selected services together, and *(c)* signal to the selected services that the workflow is composed and work should begin. Therefore, mathematically, the execution of the workflow proceeds in a similar manner to that described in Section 6.2.2 but with some additional workflow specific processing carried out by each selected node. The top level vector, *WP*, is prepared to make the initial request as per eq. (6.3):

$$\boldsymbol{WP_1} = (\boldsymbol{p_0^0} \cdot (\boldsymbol{T} + \boldsymbol{WP}))^{-1} = \boldsymbol{p_0^{-1}} \cdot \boldsymbol{T^{-1}} + \boldsymbol{Recruit_{Nodes}} + \boldsymbol{noise}$$

When multicast, this activates the $\boldsymbol{Recruit_{Nodes}}$ clean-up service, which carries out the same operation to initiate the recruitment phase:

$$\begin{aligned} \boldsymbol{Recruit'_{Nodes}} &= (\boldsymbol{p_0^0} \cdot (\boldsymbol{T} + \boldsymbol{Recruit_{Nodes}}))^{-1} \\ \boldsymbol{R'_1} &= \boldsymbol{p_0^{-1}} \cdot \boldsymbol{T^{-1}} + \boldsymbol{Z_1^0} + \boldsymbol{p_1^{-1}} \cdot \boldsymbol{Z_1^1} + \boldsymbol{p_1^{-1}} \cdot \boldsymbol{p_2^{-1}} \cdot \boldsymbol{Z_1^2} + \dots \end{aligned}$$

$\boldsymbol{Z_1}$ is a request for an mProjectPP service, which will be matched by all listening mProjectPPs. Acting as the local arbitrator, the $\boldsymbol{Recruit_{Nodes}}$ service multicasts its preferred match from the replies received. The newly discovered and activated service uses the current permutation of the $\boldsymbol{T}$ vector to calculate its position ($\boldsymbol{NodeID_r^n}$) in the $\boldsymbol{Recruit_{Nodes}}$ phase from which it can calculate its unique parent and child vector names to be used during the $\boldsymbol{Connect_{Nodes}}$ phase. Thus, the first mProjectPP, having position *p0* and being a $\boldsymbol{Z_1}$, calculates its parent and child names as:

$$\begin{aligned} \boldsymbol{\mathcal{P}_0} &= \boldsymbol{Z_1^0} \cdot (\boldsymbol{NodeID_r^0} \cdot \boldsymbol{Parent_r}) \\ \boldsymbol{\mathcal{C}_0} &= \boldsymbol{Z_1^0} \cdot (\boldsymbol{NodeID_r^0} \cdot \boldsymbol{Child_r}) \end{aligned}$$

It then enters *Listening for Connections Mode* while, as the new *local arbitrator*, it also multicasts the next recruitment request by performing an unbind using eq. (6.5) on its received vector $\boldsymbol{R'_1}$,

$$\boldsymbol{R'_2} = (\boldsymbol{p_1^{-1}} \cdot \boldsymbol{Z'_1})^{-1} = \boldsymbol{p_1^{-1}} \cdot \boldsymbol{p_0^{-2}} \cdot \boldsymbol{T^{-2}} + \boldsymbol{p_1^{-1}} \cdot \boldsymbol{Z_1^{-1}} + \boldsymbol{Z_1^0} + \boldsymbol{p_2^{-2}}.\boldsymbol{Z_1^1} +$$

The will cause another mProjectPP to be selected and this decentralized process repeats until the last service to be recruited, the Z_9, mjPeg, service unbinds and transmits the next vector, the $\boldsymbol{Recruit_{Nodes}}$ *StopVec*.

The $\boldsymbol{Recruit_{Nodes}}$ clean-up service detects its stop vector, causing it to perform an unbind, using eq. (6.5) and multicast of $\boldsymbol{WP'}$ thereby activating the $\boldsymbol{Connect_{Nodes}}$ phase:

$$\boldsymbol{WP}_2 = (\boldsymbol{p}_1^{-1} \cdot \boldsymbol{WP}_1)^{-1} = \boldsymbol{p}_1^{-1} \cdot \boldsymbol{p}_0^{-2} \cdot \boldsymbol{T}^{-2} + \boldsymbol{Connect_{Nodes}} + \boldsymbol{noise}$$

At this point all recruited services are listening for connection requests on their unique parent and child vectors. The activated $\boldsymbol{Connect_{Nodes}}$ service, acting as a clean-up service, uses eq. (6.3) to initiate and activate the first *parent* node of the $\boldsymbol{Connect_{Nodes}}$ phase:

$$\begin{aligned} \boldsymbol{Connect'_{Nodes}} &= (\boldsymbol{p}_0^0 \cdot (\boldsymbol{T} + \boldsymbol{Connect_{Nodes}}))^{-1} \\ \mathbb{P}'_1 &= \boldsymbol{p}_0^{-1} \cdot \boldsymbol{T}^{-1} + \mathbb{P}_1^0 + \boldsymbol{p}_1^{-1} \cdot \mathbb{C}_1^1 + \boldsymbol{p}_1^{-1} \cdot \boldsymbol{p}_2^{-1} \cdot \mathbb{P}_2^2 + \ldots \end{aligned}$$

When a service matches its *parent* vector, it performs the next unbind and multicast to activate its associated *child* service, automatically informing the child service of the location of its resources/output/ip-address. When a service matches to its *child* vector, it can lookup the sender's (producer/parent's) IP-address and send a unicast *hello* message to the *parent*, thus establishing the required connection before activating the next *parent* by performing a further unbind and multicast of the $\boldsymbol{Connect_{Nodes}}$ vector. This process repeats until the final child request is processed causing the $\boldsymbol{Connect_{Nodes}}$ service to detect its *StopVec*, which, in turn, causes it to unbind and multicast the *StartVec* indicating to all nodes that the workflow has been fully constructed and processing can be started.

6.2.5 Pre-provisoning and learning to get ready

From eq. (6.5) we see that each workflow step is exposed by iterative application of *p* vector permutations. Non-matching services can use this method to *peek* a vector enabling anticipatory behavior such as the pre-provisioning of a large data-set or changing a device's physical position (e.g., drones). Obviously, services can *peek* multiple steps into the future and could *learn* how early to start pre-provisioning. This ability to anticipate could be used to perform more complex on-line utility optimization learning. For example, a drone monitoring multiple workflows may be able to understand that it will be needed in 10 minutes to perform a low priority task and in 15 minutes for a high priority task. Under these circumstances, it may choose not to accept the low priority task.

6.2.6 Learning from context

As can be seen in eq. (6.4) and eq. (6.6), when a particular workflow step is exposed for discovery and execution by unbinding, it is 'surrounded by' (i.e. it is in superposition with) the rest of the workflow steps that, as can be seen, are permuted in a specific way depending on the position of the currently exposed/active service in the workflow. We can think of this as the current permutation of the workflow vector and it constitutes a semantic context for the workflow step currently in focus.

When a node participates in multiple workflows, these contexts may be usable by learning algorithms to allow a node to learn/generalise for itself what workflows it is good at. Because all workflow messages are passed by multicast, these same contexts might be usable by learning algorithms to model the meaning of workflows and sub-workflow as well as the meaning, i.e., function, of microservices that participate in workflows, in the same way that word2vec type algorithms learn the semantic similarity of words by considering the context of each word appearing in many sentences.

For example, when a service successfully participates in a workflow, it could remember the permutation state of the workflow vector in which it was activated. If the service successfully participates in the same workflow repeatedly, the workflow context memory can be used to increase the service's utility with respect to the specific workflow.

6.3 Hierarchical VSA Scaling Preserving Semantic Similarity

Simpkin et al. [4] included an empirical evaluation designed to show that the encoding scheme presented in this chapter is scaleable while preserving a measure of semantic similarity. Using sets of randomly generated vectors, a number of experiments where

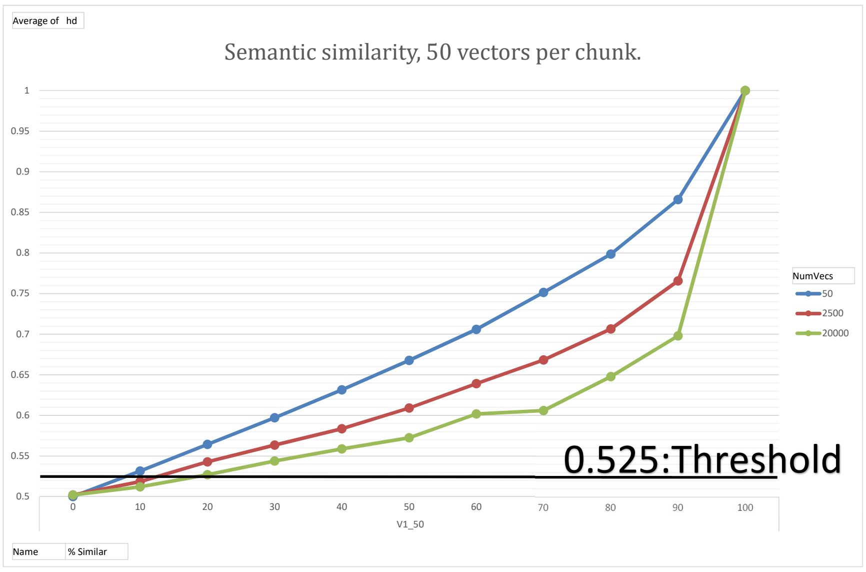


Figure 6.3: Semantic Similarity 20k vectors, Chunk Size 50.

carried out. Two sets, *set1* and *set2*, of 10kbit random vectors were generated and both sets were *chunked* using eq. (6.1). The resulting top-level vectors were then compared by measuring Hamming distance similarity. The comparison was repeated after randomly choosing an increasing percentage of vectors from set2 and copying them into the same position in set1. The Hamming distance similarity was then recalculated as the sets become increasingly more similar. The expected result was that no similarity would be detected when each set had none or very few common vectors and that Hamming distance similarity would increasingly improve as the sets become similar.

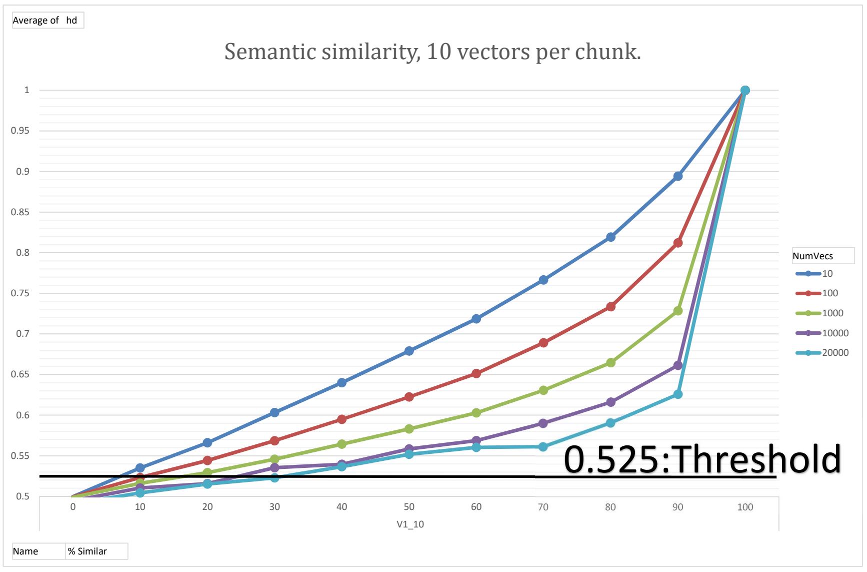


Figure 6.4: Semantic Similarity 20k vectors, Chunk Size 10

Figures 6.3 and 6.4 show the results using a set size of 20k for chunk sizes of 50 and 10. Each line shows the average similarity—i.e., $(1 - \boldsymbol{HD})$—of chunks at each level in the chunk hierarchy. For example, in Fig. 6.4, there is only 1 vector at the top level (green), 8 chunks at each point of red and 400 chunks at each point on the blue line. Comparing Figs 6.3 and 6.4 we see that ability to detect a semantic match decreases for smaller chunk size.

When the chunk size is 50, we are able to detect a match with as little as 20% similarity for the single top level vector, whereas at a chunk size of 10, we can only detect a match when similarity is approximately 30%. This is to be expected since a smaller chunk size implies more *majority-vote* operations, which means more noise is introduced. In addition, an even-numbered chunk size causes the addition of additional noise in the form of random splitting of ties during the majority-vote operation. Thus, using the largest chunk size consistent with the dimensionality of the vectors being used will facilitate better semantic matching at higher conceptual levels.

6.4 Limitations

6.4.1 Role Vector Distribution

When building SV object descriptions using the methods described in Chapter 5, service attributes values are represented by binding each vector value to an *atomic* role vector, creating a role-filler pair. For service objects to cooperate and match service requests, identical copies of these role vectors must be available to each service object. Similarly, when encoding workflow request vectors as described in Chapter 6, the workflow step '$\boldsymbol{p}$' vectors used to define a position in the workflow, see eq. (6.1), the '$\boldsymbol{T}$' vector described in Section 6.2.2 and several other role vectors must all be common to each service node so that it can operate on the workflow vector when executing a workflow.

A limitation is how one can ensure that role vectors are consistent across multiple heterogeneous compute devices? The simplest solution might be for VSA enabled sensor and service objects to generate the role vectors in a well-defined order using an agreed-upon pseudo-random number generator algorithm and seed. However, can role vectors be generated in this way on heterogeneous hardware? Are generated role vectors guaranteed to be consistent?

An alternative approach that is guaranteed to work is to maintain a centralized database of the role vectors that services can download on start-up. The role vector data set could be digitally signed and allow distributed services to cache the data set, hence saving bandwidth until it detects that the digital signature has changed when it needs to download the changed data set.

6.4.2 Clean-up services

For large workflows consisting of many hundreds of microservice steps, the eq. (6.1) encoding scheme produces a set of '*clean-up*" service vectors as described in 6.1, 6.2.1 and 6.2.2. How to most efficiently distribute these clean-up vectors throughout the network must be taken into consideration. This issue was simplified greatly during the evaluations described in Chapter 9 because the network could be created with clean-up services in place when it was instantiated. The problem is particularly acute when new complex workflows are created and need to be launched into an existing network. A future work objective is to consider how best to disseminate clean-up service vectors into an existing network. One approach is to allow the vectors to diffuse out from the requester that first issues the workflow request (the workflow creator has a full copy of all the clean-up service vectors). Figure 8.3 shows that, in the current implementation of the VSA cognitive layer's Listener, the workflow request vectors are cached. The cached vectors can be used to allow any node to act as a clean-up service for vectors that it has previously cached. Another approach might be to use an announcement scheme (prior to the workflow being needed in anger). This scheme would consume additional bandwidth if many new workflows for which no existing sub-workflow clean-up vectors exist are constantly being created.

6.4.3 Slot encoding

In consideration of eq. (6.2), we can see that it is effectively a slot encoding scheme, which are known to be brittle when comparing objects that are even slightly out of sync. This means that if we encode two sentences, for example,

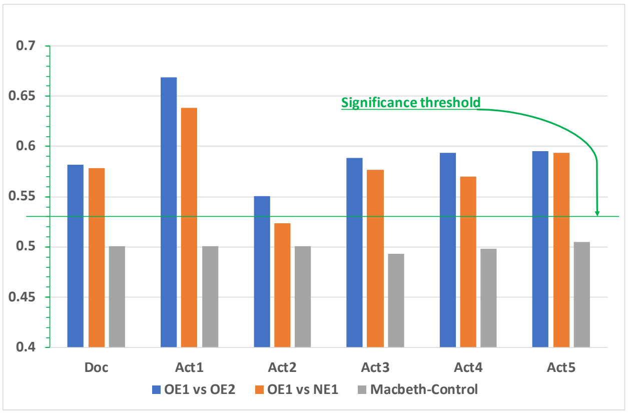
$$\boldsymbol{S_1} = \boldsymbol{p_0^0 \cdot My^1 + p_0^0 \cdot p_1^0 \cdot name's^2 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot Jim^3} \tag{6.8}$$

$$\begin{aligned}\boldsymbol{S_2} &= \boldsymbol{p_0^0 \cdot Hello,^1 + p_0^0 \cdot p_1^0 \cdot my^2 + p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot name's^3} \\ &\quad \boldsymbol{+ p_0^0 \cdot p_1^0 \cdot p_2^0 \cdot p_3^0 \cdot Jim^4},\end{aligned} \tag{6.9}$$

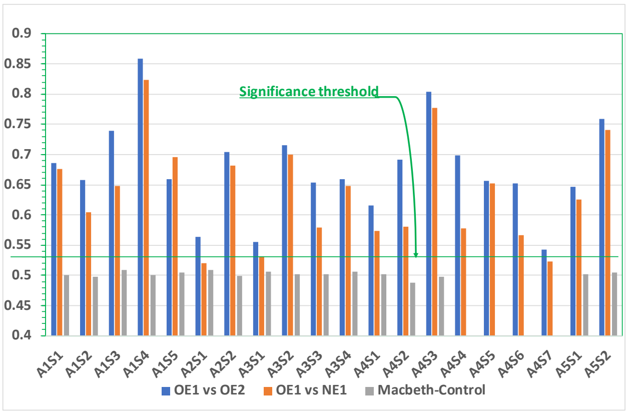
then $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$ would give a poor HDS match because all but the first term is permuted differently and each term after *'The'* is orthogonal to their counter parts in the other sentence. Nevertheless, the scheme can prove surprisingly successful when the data is segmented frequently using chunking. For example, if we are encoding a book and we use a variable chunk size that matches the natural punctuation breaks then there are multiple opportunities to re-sync. The wording of a particular sentence may vary which would damage the comparison for such a sentence; however, the story line is expected to follow and so subsequent sentences may be very similar.

To illustrate how the semantic matching works, Figure 6.5a shows a HDS comparison between vectors generated from different editions of the play Hamlet. The comparisons are made between two ‘Old English’ (OE) versions of the play (blue histogram in the figures) and comparisons between an OE and a modern ‘New English’ (NE) version of the play (orange histogram). Also shown is a comparison with vectors generated from the play Macbeth (grey).

The results show that at the level of the entire play and for all acts, the two OE versions are semantically the same. Perhaps, surprisingly, the comparison between the OE and NE versions also shows that these are also semantically similar except for Act 2, which



(a) Hamming Similarity between different versions of Hamlet at the level of the entire play and for the individual Acts.



(b) Hamming Similarity between different versions of Hamlet at the level of the individual Scenes.

Figure 6.5: Old English Hamlet compared to New English Hamlet.

does not match. Comparison with the Macbeth vectors shows that these two plays are essentially uncorrelated. Similar comparisons can also be made at the lower semantic levels and Figure 6.5b shows the equivalent comparisons at the level of the scenes of the play. At the scene level, the OE1 vs OE2 are all semantically similar. The OE vs NE comparison is below the threshold for Act2 Scene1, Act4 Scene7 and is marginal for Act3 Scene 1, but otherwise the versions are still semantically similar.

While the eq. (6.1) encoding scheme might work well for complex workflows consisting of many workflow steps, the ability to detect similarities between vectors describing short workflow sequences will be limited. A future work objective is to allow for variations in workflow sequence structure by using a vector weighting, or '*shaping*', scheme that '*bleeds*' the previous and subsequent workflow steps into the current/target workflow step. For example, using eqs. (6.8) and (6.9), consider the second slot in each; using a weighting scheme, '*name's*', in eq. (6.8) would contain a weak representation of both '*My*' and '*Jim*' and similarly in eq. (6.9), '*name's*' will contain weak representations of its surrounding words. Hence, when comparing workflow vectors that are out of sync by a small number of steps, better comparisons could be obtained.

6.5 Summary

This chapter describes a new recursive VSA bundling and binding scheme that is fully recursive without the issues that plague previous bundling and binding schemes and scales to a practically unlimited number of sub-vectors via chunking. It can be used to encode ordered sequences of workflow steps which is extended to the encoding of DAG workflows in Section 6.2.4. The encoding creates a set of VSA vectors representing a hierarchical chunk tree where the root and non-terminal nodes act as proxy *clean-up* services that create an instantiation pathway to each real workflow service request vector located at the terminal nodes. When real worker services are activated by matching to an exposed workflow request step, they can interrogate their activation position and decode workflow metadata via a known role vector, the $\boldsymbol{T-Vector}$. To discover and execute a workflow, the workflow vectors and sub-workflow vectors (*clean-up services*) are repeatedly unbound and passed from peer-to-peer using multicast so that listening services can match to the newly unbound workflow step (service description request). The end of a workflow request vector is signaled via a $\boldsymbol{StopVec}$, role vector.

The workflow encoding supports pre-provisoning because it is very simple to traverse a workflow request vector exposing the individual workflow request steps. This means that, on receipt of a multicast workflow request, services can simply wind forwards (unbind) one or more steps to see if they match to any of the subsequent steps. (Note that it is just as simple to unbind in reverse, for example, to inspect the workflow's metadata.)

Every step within a workflow vector has a unique permutation, which might allow service agents to learn about the function of both services (analogous to words in an NLP application) and workflow request vector similarity (analogous to sentence similarity in NLP).

Chapter 7

A Mathematical Model for VSA Vector Truncation

This chapter describes how BSCs behave like digital holograms, allowing VSA workflow orchestration messages to be truncated without loss of information (R3). This reduces bandwidth overhead; an important consideration in low-bandwidth environments. The levels of similarity between vectors we would like to decode (e.g., activating the correct sensor in preference to some similar sensor) impose a limit on this. A mathematical derivation of the minimum number of bits needed to differentiate between similar BSC vectors is given, including an empirical verification of the derived mathematical model.

7.1 Holographic properties of Binary Spatter Codes

This work was originally presented in Simpkin et al. [5]. In Section 4, we explained that the symbolic vectors we use are typically 10,000 bit vectors. However, one of the important properties of large BSC compound vectors is that they are distributed representations of the bundled sub-vectors. As such, if the number of sub-vectors is small, then successful decoding and matching comparisons can be made even when the vectors are truncated to a shorter length. This holographic property of BSCs suggests an approach for saving bandwidth on the message payloads that are exchanged over the communications network. Essentially, vectors can be truncated without affecting the VSA bindings and comparison performance. In this section, we consider the mathematical basis for such a scheme and later, in Section 9.2.6, we describe how the scheme is used in practice and the typical network bandwidth savings that are possible.

When multiple VSA sub-feature vectors are combined using *majority_vote* addition, the resultant vector is a single VSA vector of the same size as its sub-features and represents the set of sub-features. This can be an ordered set; for example, a workflow composed via Eq. 6.1, or an un-ordered bag of role-filler pairs used to encode a service description vector, as described in Section 5.1. Such compound vectors might be thought of as a representation of the *concept* implied by the collection of sub-feature vectors be it ordered, in the case of a workflow,

e.g., track_car might be the name given to a high-level workflow vector that defines the steps required to track a vehicle, or unordered, such as a *person_record*. In VSA, these types of compound vectors are commonly called *chunk* vectors, and the number of sub-feature vectors in a chunk is its *ChunkSize* [40, 110]. An important property of such chunk vectors is that the distribution of 1s and 0s remains random. This is because chunks are ultimately made from random *atomic* vectors at the bottom of the chunk hierarchy. Note that when bundling via Eq. 6.1, the permutation vectors ensure that each sub-feature vector is orthogonal to the other vectors in the sum; however, after unbinding, the exposed sub-feature vector will bear similarity to other SV descriptions.

As explained in Section 4.3, the *majority_vote* addition operator creates a *superposition* of sub-vectors such that each sub-vector is represented by many, but not all, binary bits of the resultant vector and each sub-vector has a unique random distribution of bits within the resultant vector. Hence, a BSC chunk vector can be considered to be a digital analogue of a hologram in that each sub-vector is represented equally by any reasonably long sub-segment of the original vector. For example, the mean normalized hamming similarity between a chunk vector $\boldsymbol{S} = [\boldsymbol{A} + \boldsymbol{B}]$ and its sub-features is $\boldsymbol{HSim(S, A) = HSim(S, B) = 0.75}$. This means that 75% of the bits in $\boldsymbol{S}$ will match to $\boldsymbol{A}$ and a different 75% of the bits in $\boldsymbol{S}$ will match with $\boldsymbol{B}$. If we compare $\boldsymbol{S}$ to a random vector not in $\boldsymbol{S}$, then it is easy to see that 50% of the bits will match. Thus, 25% of the bits in $\boldsymbol{A}$ (or $\boldsymbol{B}$) actively differentiate it from a random vector. If we take any reasonable length segment of $\boldsymbol{S}$, for example, the first 1000 bits, and perform a hamming similarity comparison to $\boldsymbol{A}$ (or $\boldsymbol{B}$), we will get approximately, the same 0.75 result. Hence, if there is only a small number of sub-vectors in a workflow request or reply vector, then we can truncate the vector before transmission without affecting workflow operations. On *'seeing'* a truncated vector, the VSA service simply truncates its vector description to the same length and performs its operations in the normal way.

The limit to which a chunk-vector can be truncated whilst maintaining the ability to detect the sub-vectors it contains is directly related to the number of sub-vectors bundled. The degree of similarity between sub-vectors in the known vocabulary/clean-up memory and the size of the vocabulary are also important factors in determining the minimum size to which a chunk vector can be truncated. When matching to a truncated vector, it is necessary to be able to distinguish between the target sub-vector embedded in the chunk in preference to some similar sub-vector that is not part of the chunk. Truncating a chunk vector to its minimum decodable size is a direct corollary for the capacity of a chunk for which estimations and lower bounds are derived in [36, 46, 110]. Below, we provide a mathematical derivation of the minimum number of bits needed to differentiate between BSC vectors when similarity between vectors is expected to exist in the network (e.g., similar sensor objects).

Consider the following scenario: we compose a workflow that aims to activate a camera sensor '$\boldsymbol{Y}$' as its first step. There are many camera sensors in the system, and we know that there are cameras of similar type in other locations. Suppose some other camera-service sensor '$\boldsymbol{X}$' elsewhere in the network, differs from the target camera-service in only three parameters: the frame rate, resolution and location. If the total number of sub-feature vectors used to describe each camera is 9, then the two cameras share 6 common sub-features, making them notionally 66% similar to each other. Let $\boldsymbol{WF}$ be a workflow request for camera $\boldsymbol{Y}$, eq. (7.1), which is activated by unbinding the workflow vector $\boldsymbol{WF}$ exposing $\boldsymbol{Y'}$ as a noisy copy of itself, $\boldsymbol{Y'} = \boldsymbol{Z_1}$. We want to determine the minimum number of bits that can be used for transmission of the unbound workflow vector $\boldsymbol{WF'} = \boldsymbol{Z_1}$ so that camera $\boldsymbol{Y}$ will be matched and differentiated in preference to all other service vectors in the system.

$$\boldsymbol{WF'_1} = \boldsymbol{Z_1} = \boldsymbol{p_0^{-1}} \cdot \boldsymbol{T^{-1}} + \boldsymbol{Y^0} + \boldsymbol{P_1^{-1}} \cdot \boldsymbol{Z_2^1} + \boldsymbol{p_1^{-1}} \cdot \boldsymbol{p_2^{-1}} \cdot \boldsymbol{Z_3^2} \tag{7.1}$$

Consider each service $\boldsymbol{Y}$ and $\boldsymbol{X}$. On receipt of $\boldsymbol{Z_1}$, both will perform a hamming similarity calculation to check for a match. Let the match values be μ_y and μ_x respectively. The target service, $\boldsymbol{Y}$ is a sub-vector of $\boldsymbol{Z_1}$, therefore, the mean normalized hamming similarity, μ_y, is calculated as described in Section 4.3 and is given by:

$$\mu_y = \frac{1}{2^m} \sum_{i=\lfloor m/2 \rfloor}^{m} \binom{m}{i} \begin{cases} m = n & if\ n\ even. \\ m = n - 1 & if\ n\ odd. \\ n\ = \#\ of\ vectors. \end{cases} \tag{7.2}$$

Since we know the similarity of $\boldsymbol{Y}$ to $\boldsymbol{Z}$, we can calculate the similarity of $\boldsymbol{X}$ to $\boldsymbol{Z}$ if we can find the similarity of $\boldsymbol{X}$ to $\boldsymbol{Y}$, that is $\mu_{sim\_xy} = \boldsymbol{HSim}(\boldsymbol{X}, \boldsymbol{Y})$. The similarity of service vector $\boldsymbol{X}$ having sub-vectors in common with $\boldsymbol{Y}$ is then given by:

$$\boldsymbol{P}(\boldsymbol{X} = \boldsymbol{Z}) = \boldsymbol{P}(\boldsymbol{X} = \boldsymbol{Y}) * \boldsymbol{P}(\boldsymbol{Y} = \boldsymbol{Z}) + \boldsymbol{P}(\boldsymbol{X} \neq \boldsymbol{Y}) * \boldsymbol{P}(\boldsymbol{Y} \neq \boldsymbol{Z}) \tag{7.3}$$

That is, (the probability that $\boldsymbol{X} = \boldsymbol{Y}$ when $\boldsymbol{Y} = \boldsymbol{Z}$) + (the probability that $\boldsymbol{X} \neq \boldsymbol{Y}$ when $\boldsymbol{Y} \neq \boldsymbol{Z}$), or in terms of mean hamming similarities:

$$\mu_x = \mu_y * \mu_{sim\_xy} + (1 - \mu_y) * (1 - \mu_{sim\_xy}) \tag{7.4}$$

Referring to Figure 7.1, since $\boldsymbol{X}$ and $\boldsymbol{Y}$ are I.I.D strings of 1s and 0s we can consider a single bit column in order to calculate μ_{sim_xy}. Consider any column in Figure 7.1, say the left most column; there are three shared vectors, which means there are 2^3 possible combinations of bit values, $\{000, 001, \cdots, 111\}$. Equation (7.6) gives the probability,

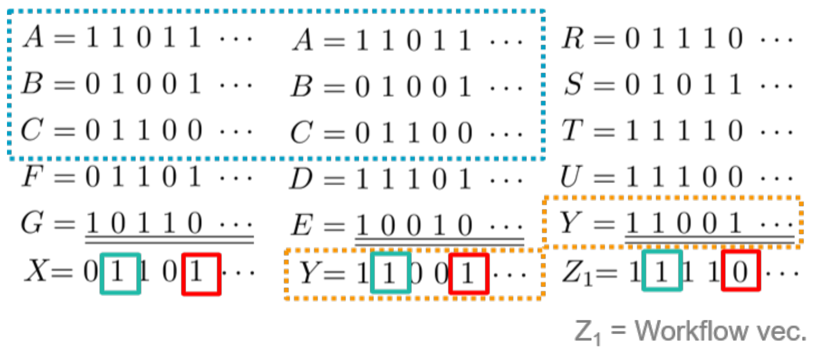


Figure 7.1: Shared vectors contribute to the similarity between similar service-vectors.

P_i, of the i^{th} bit pattern appearing in a column among the shared vectors. For example, the pattern $\{111\}$ will only occur with probability $1/2^3 = 0.125$, whereas the equivalent patterns $\{001, 010, 100\}$ (order does not matter for *majority_sum*) will occur with probability $3/2^3 = 0.375$. Equations (7.7) and (7.8) give the probability that the i^{th} shared bit pattern will determine the the value of the corresponding bit in X and Y. The mean hamming distance between X and Y is then given by[1]:

$$\mu_{sim\_xy} = \sum_{i=(S+1)\text{div}2}^{S} P_i * (x_i * y_i + (1 - x_i) * (1 - y_i)) \tag{7.5}$$

where

$$P_i = \frac{1}{2^S} * \frac{2}{2^{(S \text{ div } i)} - 1} \binom{S}{i} \tag{7.6}$$

and

$$x_i = \frac{1}{2^{N-S}} * \sum_{j=W_N} \binom{N-S}{j} \tag{7.7}$$

[1]Note, in the original paper [5] the calculation of unknown, *similar*, expected values (referred to in the paper and below as μ_x) was obtained empirically.

and

$$y_i = \frac{1}{2^{M-S}} * \sum_{j=W_M}^{M-S} \binom{M-S}{j} \tag{7.8}$$

Y	$=$ The target vector to be matched.
X	$=$ A similar vector to Y within the 'service' vocabulary.
S	$=$ Number of vectors common to both X and Y.
M	$=$ Total number sub-vecs in Y.
N	$=$ Total number sub-vecs in X.
P_i	$=$ the probability of the i^{th} shared bit combo occurring. Note, when S is even, $\frac{2}{2^{\lfloor S/i \rfloor - 1}} = 1$ for the first term in $i = \{\dots\}$ and 2 for all terms otherwise.
x_i	$=$ the probability that the bit in X will match the 'sense' (1 or 0) of the majority taken from the shared vecs S, $P(X_n = S_{maj} \vert i_{combo})$.
y_i	$=$ same as x_i, but for Y, $P(Y_n = S_{maj} \vert i_{combo})$.
W_N	$=$ the number of bits needed to make majority in X, given i bits from the shared vectors, $W_N = max(\lfloor N/2 \rfloor + 1 - i,\ 0)$.
W_M	$=$ the number of bits needed to make majority in Y, given i bits from the shared vectors, $W_M = max(\lfloor M/2 \rfloor + 1 - i,\ 0)$.
$N - S$	$=$ number of bits available from X's sub-vecs that aren't shared.
$M - S$	$=$ number of bits available from Y's sub-vecs that aren't shared.

Figure 7.2 compares eq. (7.5) to empirical measurement of μ_{sim_xy}. The number of vectors in the target service-vector, Y, was fixed at 35. The total number of vectors in the similar service-vector, X was allowed to vary around $35 \pm 33\%$ to show that the calculation is accurate even when X and Y have a differing numbers of sub-vectors. The X-axis is the number of shared vectors, and the Y-axis is the hamming similarity of both calculated and empirical findings. The empirical test was conducted using 100 million trails for each data point. From the graph, it can be seen that the theoretical curve is coincident with the empirical result, this is because results agreed to better than 4 decimal places. To try to give a sense of how well the results match and highlight

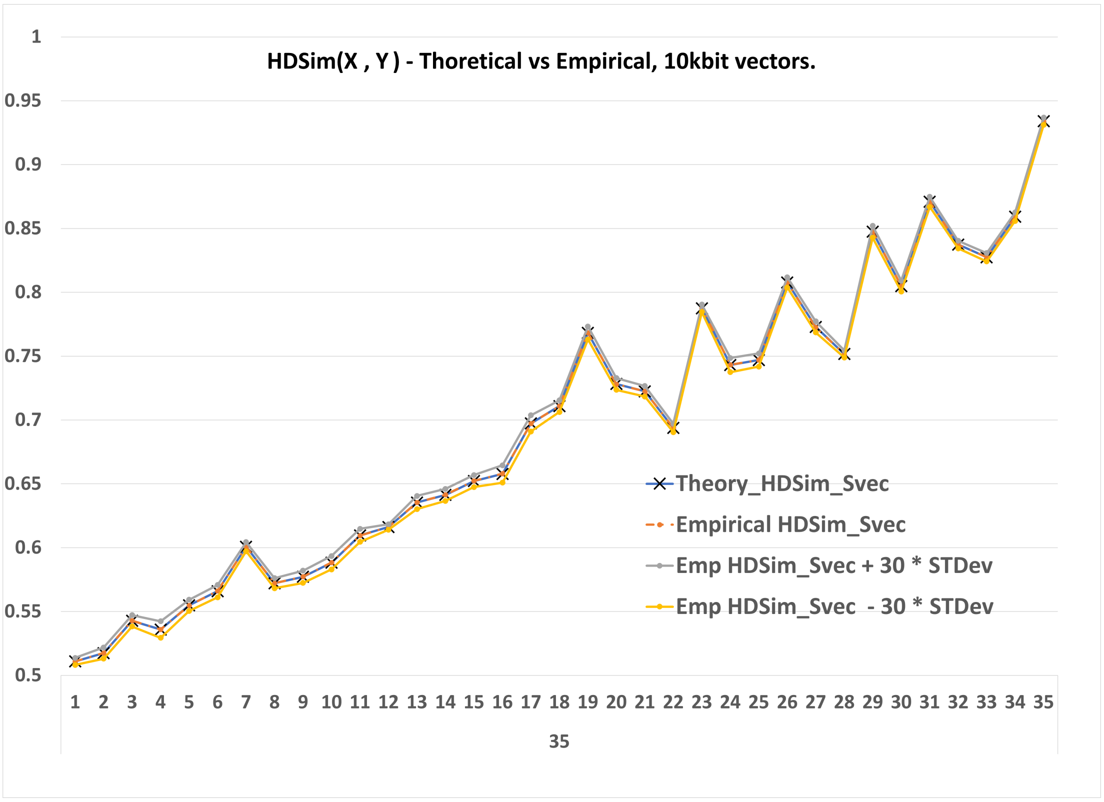


Figure 7.2: Comparison of the theoretical similarity between vectors, eq. (7.5), and an empirical evaluation.

any slight deviations, the empirical result was translated up and down by 30 standard deviations; see the grey and pale-orange lines.

Now that we can accurately predict the similarity between vectors having a number of sub-feature vectors in common, we can use eq. (7.3) to predict μ_x, the mean similarity between a competing service-vector and the noisy workflow-request vector Z_1. Figure 8.4 presents correlation graphs for eq. (7.4) compared to empirical measurement for two vector dimensions and shows that the results obtained are highly accurate.

For the empirical measurements, a set of random vectors was repeatedly generated and bundled into a target vector Y. A proportion of the same set, along with other random vectors, was then used to generate a *'competing'* similar vector X. A multi-step workflow vector was simulated by bundling Y with a number of other random

$$\mu_x = \mu_y * \mu_{sim\_xy} + (1 - \mu_y) * (1 - \mu_{sim\_xy})$$

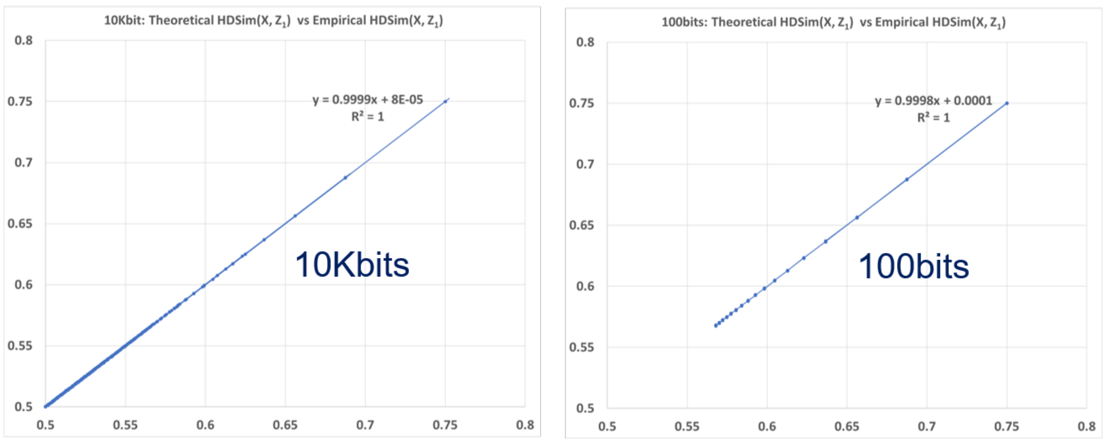


Figure 7.3: Comparison of the theoretical similarity between vectors, eq. (7.5), and an empirical evaluation.

vectors as shown in the right-hand side of Figure 7.1. The similarity between X and Z_1 was then measured and compared to the result obtained from eq. (7.4). As in the previous test, 100 million trials were carried out. For completeness, Figure 7.4 shows correlation graphs for the target vector's normalized mean hamming similarity, μ_y, calculated using eq. (7.2). These graphs were obtained during the same test.

7.2 Truncation

Note, in the original paper [5], the continuous normal distribution and error function were used to estimate the joint probability and calculation of minimum truncation length. This is improved upon below by use of the discrete binomial probability to calculate on the overlapping distributions.

All Hamming distance/similarity matching operations carried out by VSA services will follow a binomial distribution centered on their expected values, see Figure 7.5. Having established the mean normalized hamming similarity (expected value) between the workflow-request vector Z_1 and the target service-vector $Y = \mu_y$, as well as other

$$\mu_y = \frac{1}{2^m} \sum_{i=\lfloor m/2 \rfloor}^{m} \binom{m}{i} \begin{cases} m = n & if\ n\ even. \\ m = n - 1 & if\ n\ odd. \end{cases} \quad n\ = \#\ of\ vectors.$$

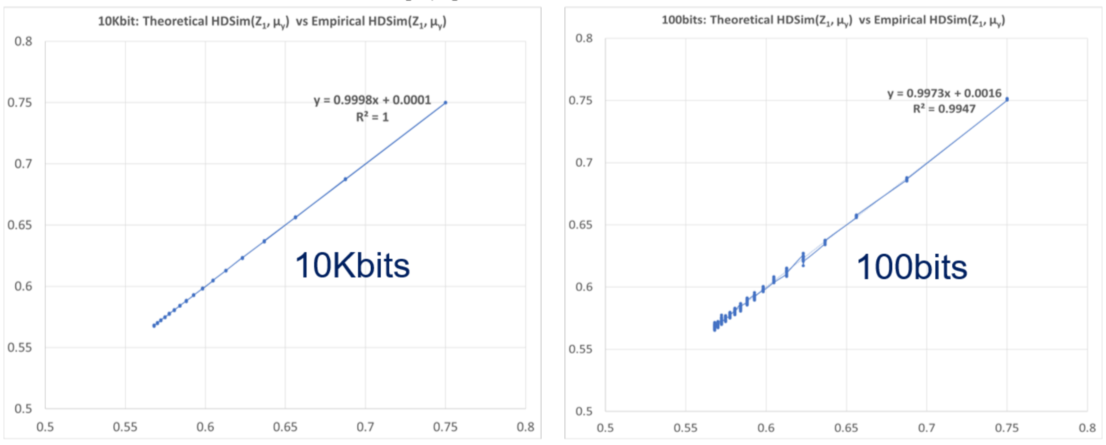


Figure 7.4: Comparison of the theoretical similarity between vectors, eq. (7.5), and an empirical evaluation.

potentially similar service-vectors $X = \mu_x$, we can now calculate the minimum number of bits required to successfully match Y in preference to similar vectors X. Let,

$$P_Y \sim B(i, \nu_y), \quad 0 \leq i \leq D \tag{7.9}$$

and

$$P_X \sim B(i, \nu_x), \quad 0 \leq i \leq D \tag{7.10}$$

Where:

P_Y	$=$ The discrete probability distribution of $HD(Y, Z_1)$
P_X	$=$ The discrete probability distribution of $HD(X, Z_1)$
B	$=$ The Binomial probability mass function.
ν_y	$= 1 - \mu_y$, the expected value (normalized hamming distance) of Y when compared to Z_1.
ν_x	$= 1 - \mu_x$, the expected value (normalized hamming distance) of X when compared to Z_1.
D	$=$ The vector dimension.

After T trials, the expected distribution of $HD(Y, Z_1)$ and $HD(X, Z_1)$ are given

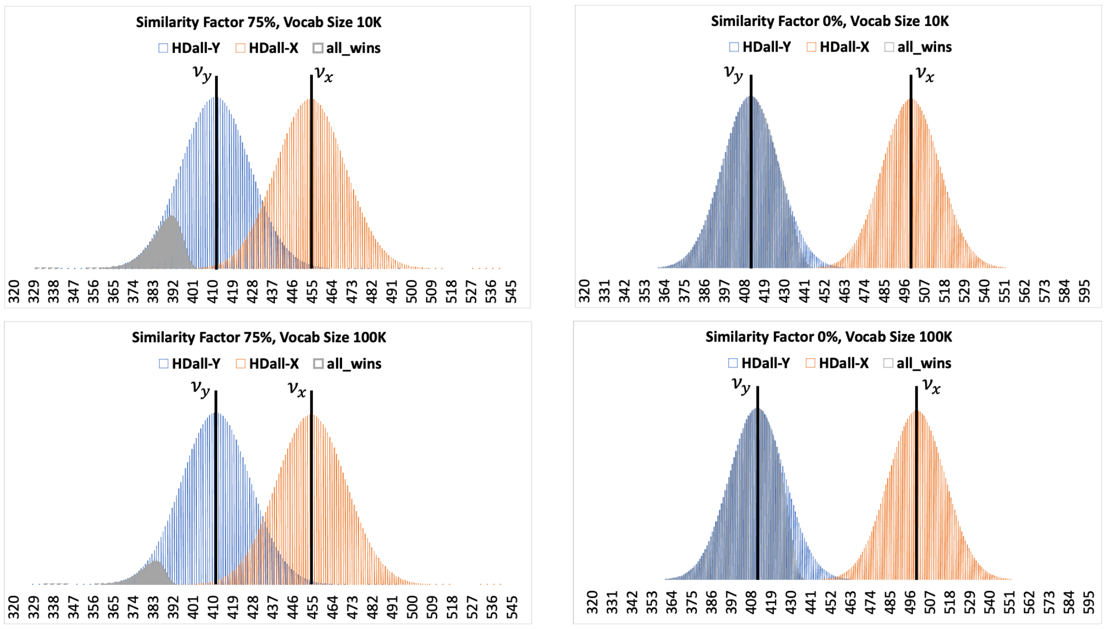


Figure 7.5: Overlapping, un-normalized, Hamming Distance distributions using different 'similarity factors' and total vocabulary sizes for vectors having $chunk\_size = 21$ and $vec\_dim = 1K$. The grey areas indicate the probability of successfully differentiating the target vector from other 'competing' vectors with a probability of error of, $P\_error \leqslant 10^{-4}$.

by (see Figure 7.5):

$$HD\_allY \sim \lfloor P_Y * T \rfloor \tag{7.11}$$

$$HD\_allX \sim \lfloor P_X * T \rfloor \tag{7.12}$$

To compute the probability of correctly matching Y to Z_1 in preference to a library of other vectors having size V, we must compute the probability of not matching with any of the alternate vectors X. The probability of not matching with any X is given by the complement of the cumulative binomial probability distribution of X:

$$P_V \sim \left(1 - \sum_{i=0}^{D} P_X\right)^V \tag{7.13}$$

Hence the probability of matching the unbound vector with the clean vector and not a noise vector is given by:

$$P_{match} \sim P_Y * P_V \tag{7.14}$$

The binomial distribution for the number of matches (i.e., wins, depicted by the grey distributions in Figure 7.5) after T trials is given by:

$$all\_wins \sim \lfloor P_{match} * T \rfloor \tag{7.15}$$

And the probability of service Y being chosen in preference to X is given by:

$$P\_win = \left(\sum_{i=0}^{D} all\_wins \right) / T \tag{7.16}$$

The probability that an alternate vector could be detected in preference to the target vector is then:

$$P\_error = (1 - P\_win) \tag{7.17}$$

In order to calculate the minimum number of bits that can be used to avoid such an error we must specify an acceptable $error\_rate$ and then find the minimum vector dimension D that satisfies this criterion:

$$min\_bits = \min \left\{ D \in \mathbb{Z}^+ \mid P\_error \leqslant error\_rate \right\} \tag{7.18}$$

Figure 7.6 shows the minimum number of bits that are required to ensure that the unbound workflow-request vector will be correctly matched with the corresponding service vector, in cases where all the service vectors are orthogonal (*i.e.,* 0% similarity) and for different levels of '*similarity factor*' between Service Vector (SV) descriptions in the network. The term '*similarity factor*' is introduced because services build their SVs autonomously, hence the level of similarity between services can only be estimated. As can be seen, when there are similarities between SVs, the minimum bits required increases. The best truncation/compression is obtained when all SVs are orthogonal.

As stated in Section 7.1, the minimum bits calculation is a corollary of the maximum capacity of BSCs. Therefore, eq. (7.18) can be used to calculate the maximum sub-vector capacity for a given vector dimension, vocabulary size and acceptable error rate. From Figure 7.6, it can be seen that the capacity of orthogonal 10kbit BSCs is 93. This concurs well with the result obtained in [46, Paper B, page 80] where Kleyko finds the capacity of orthogonal 10kbit BSCs to be approximately 89.

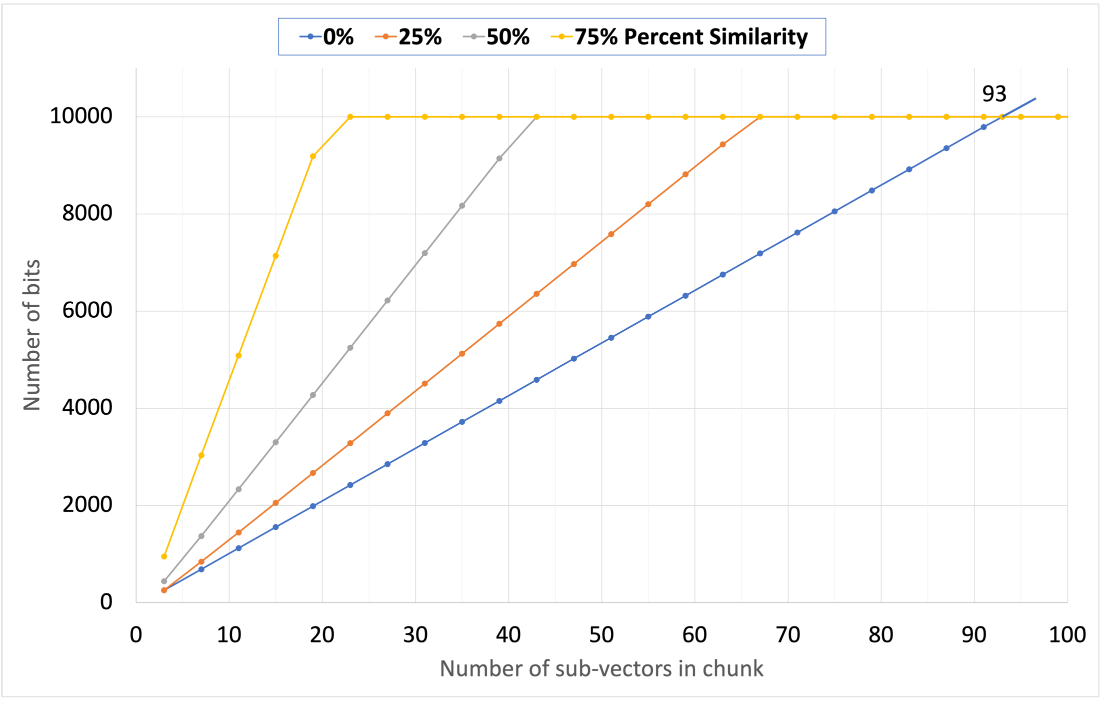


Figure 7.6: Minimum message size (No Bits) needed at different chunk sizes (number of bundled sub-vectors) and *similarity factors*. The vector dimension is capped at 10Kbit to highlight the maximum number of sub-vectors that can be encoded at this value when similarity between concept vectors is taken into account.

The model depicted by Figure 7.6 can now be used to minimize the message size of vectors transmitted to the network. Before a request or response vector is transmitted, using its knowledge of the bundled vector count, the sender simply looks up the minimum bits required to ensure that the message will be decoded correctly and truncates it accordingly. However, since the variance of HD and HDS match calculations var-

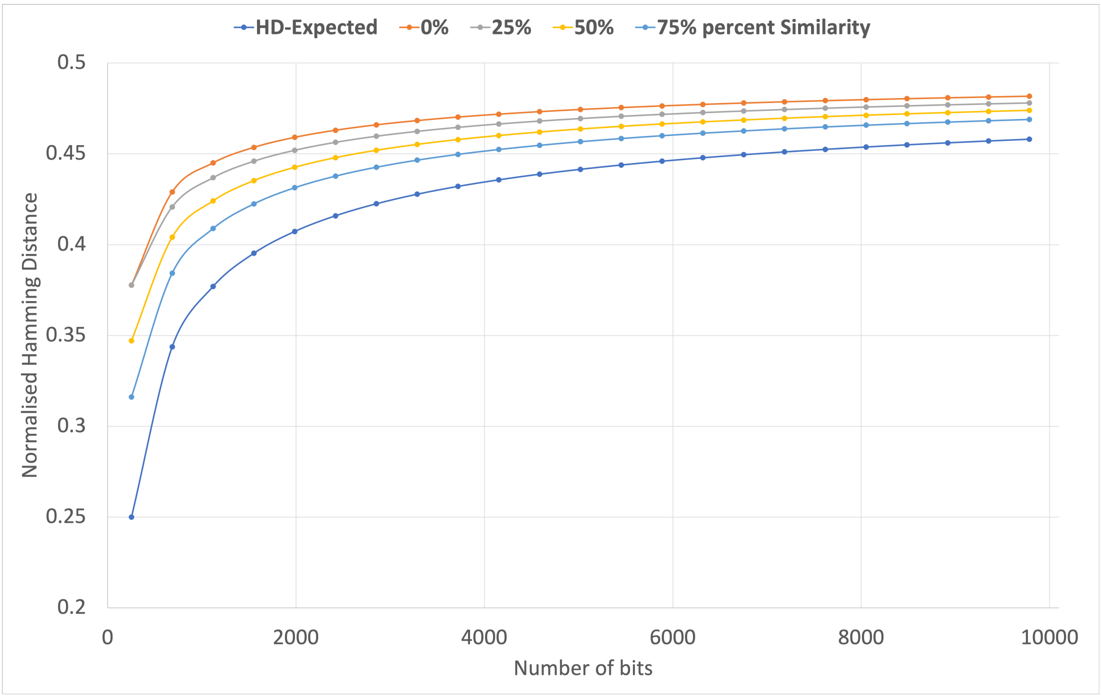


Figure 7.7: Required threshold to ensure best matching performance as a function of the vector size (No of bits) and for varying *similarity factors* .

ies with vector dimension as $1/\sqrt{D}$ and is proportional to expected value, eq. (4.3), these factors must be taken into account by VSA enabled services when calculating the match thresholds that should be used on receipt of a truncated vector. For single HD comparisons using an error probability of 10^{-6}, Figure 7.7 shows the upper bounds need to minimize both false positive and false negative activations. The blue curve is the expected value (normalized HD) and is determined from the received vector's sub-vector count using eq. (4.5). The sub-vector count can be encoded as metadata by the sender or deduced from the received vector's length using Figure 7.6. (Alternatively, the sender can calculate and embed the match threshold as metadata.) The orange curve is the upper bound HD threshold value above which matches are considered invalid. This occurs in the case when all concept vectors can be considered orthogonal because from Figure 7.6, maximum truncation (i.e., lowest vector dimension) is used in this case. When, as is expected, the distributed services and sensors bear similarity to other services in the network, a larger dimensionality is required to avoid confusion. Therefore, as shown in Figure 7.7, tighter bounds are required when there is a need to differentiate between similar service vector descriptions because vector dimension dominates eq. (4.3). For example, if the expected *similarity factor* between services in the network is $\mathbf{50\%}$ and a vector of length 439 bits is received, then from Figure 7.6, yellow curve, this implies a maximum sub-vector count of 3 vectors and from 7.7, yellow curve, the match threshold to be used to test for a valid match is $\boldsymbol{HD = 0.3471}$ or $\boldsymbol{HDS = 0.6529}$.

7.3 Limitations

The limitation with respect to using this vector truncation model is in the estimation of similarity factor. For heterogeneous service and sensor resources, the level of similarity between items that we would prefer to differentiate (in contrast to those that we would happily consider equal) is likely to be difficult to estimate. It may be possible to learn a network's similarity factor over time, and indeed, for networks that are continually evolving, the network's similarity factor could be continually updated via some form of on-line distributed learning.

7.4 Summary

This chapter described how to leverage the holographic properties of BSCs to save bandwidth for VSA workflow orchestration messages based on content. It provides a definitive mathematical treatment for calculating the minimum vector dimension needed to avoid confusion when trying to differentiate between BSC vectors that are similar. The equations described enable correct prediction of the minimum bits required to store or transmit BSC vectors in any field where BSCs might be applied, including cognitive modelling and, in our case, distributed service orchestration. The same equations can also be used to accurately predict the maximum sub-vector bundling capacity of BSCs for a given error rate and level of expected similarity between BSC vectors in an application. An experiment used to test the type of bandwidth savings that can be achieved using the vector truncation model described is given in Section 9.2.6.

Chapter 8

The VSA Platform

Building from the theoretical, empirical evaluations and implementations described in chapters 5, 6 and 7, this chapter describes how the different components are brought together into a VSA platform that has been used to address the various use cases described in Chapter 9. This chapter also addresses some of the nuances that are needed in a system that applies the VSA approach to real-world use cases, including systems considerations, message listener and buffering, memory requirements, comparison, reasoning and local arbitration to minimize the local network overhead in the service selection process.

8.1 Computational Model

For a system to behave cognitively, it must be aware of events occurring within the environment in which it operates. In other words, services must make local decisions on local data and act autonomously from the other services in the network. My computational model facilitates this in two ways:

- A one-to-many communication style, namely multicast, is used for all workflow discovery and orchestration messages (R4).
  - A delayed response algorithm, inversely proportional to match quality, minimizes network congestion (R5).

- To ensure that all objects '*speak*' a common language, a VSA is used for both workflow and service descriptions.
  - Service objects self-describe their functionality and QoS by maintaining a VSA Service Vector (SV) description (R1).
  - Multi-step linear or DAG workflow vectors, also encoded in VSA, are updated to reflect the current workflow state and passed around from peer to peer (R2).

These features create a distributed network environment within which individual services can interrogate VSA encoded workflow messages, compute their compatibility to a particular workflow step and offer service if their match quality is high enough. If a service calculates a weak match, it will delay its reply (proportional to the inverse of its match quality) and listen for better quality matches from other services on the multicast channel. Should a matching service '*hear*' a better response, it will suppress its response, saving bandwidth (R5).

This workflow paradigm is analogous to a group of humans listening to various work requests and deciding for themselves whether they are capable and available to do a piece of work. All humans can understand the work request message, and each human knows for themselves if they are available and capable of doing the work. Naturally, more than one person may offer to do a particular job, and therefore, a negotiation, spoken or unspoken, must take place to decide who gets to do the work. Interestingly, when doubtful about their ability to help, a human will often delay their offer of help (perhaps hoping that someone more qualified will step in) before stepping in themselves should no other offer materialize.

8.2 Key features

Figure 8.1 highlights some of the key features required to build the test framework which are described below.

8.2.1 Message Listener and buffer

The VSA enabled service has a capability to listen to the transmission of vectors from other services (e.g., in a multicast group) and store these messages into a temporary buffer. The decision to store a message in the buffer may require the received message to be compared with one or more vectors in the VSA memory using the Comparator component. An example of this would be the typical case of a service that only responds to semantic vectors that semantically match the specific service description vector.

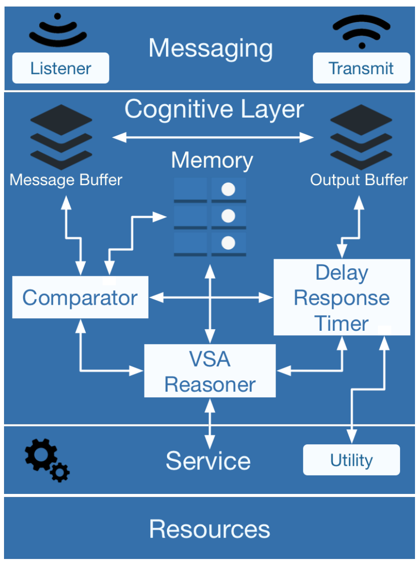


Figure 8.1: VSA Platform components.

8.2.2 Symbolic Vector Memory

The Symbolic Vector Memory is used to store vectors that are to be used for any operation required by the VSA layer of the specific service. This includes the service description vector, stop vector and various role vectors used to support vector binding and unbinding operations or to support ‘clean-up services’ that is described in Sections 6.1 and 6.2.2. In other examples, the memory is used to store application vectors that when received on previous occasions resulted in the service being selected. These vectors essentially represent the context in which the service was historically invoked, and these can be used to increase the utility (i.e., QoE) of the service if the same workflow is requested at a later time.

8.2.3 Comparator

To semantically compare vectors, we use a Normalised Hamming Similarity (HDS) measure and declare a match if the HDS is within particular ranges. The comparator uses the computed HDS to determine an appropriate time delay based on the degree of the semantic match. The semantic match time delay, t_{sm}, must be normalized to the expected value of a perfect match and is given by:

$$t_{sm} = \begin{cases} 0, \ if \ h\_match > e\_match \\ n\_latency * (e\_match - h\_match)/sim\_range, \quad otherwise \end{cases} \tag{8.1}$$

Where:

- h_match: is the similarity between the request vector and the service description vector.
- e_match: is the expected value of a perfect match calculated from the number of vectors stored in the workflow command vector using eq. (4.5).
- $n\_latency$ is a multiple of the estimated network latency. This causes the timing of responses to effectively be quantized in terms of the delayed response behaviour, section 8.2.5, because transmissions in intervals smaller than the network latency will not be seen in time for a service to suppress its response.
- $sim\_range = e\_match - 0.53$: The expected range, 0.53, is the base threshold of a valid match and represents a match probability of error of 10^{-9}.

8.2.4 VSA Reasoner

The VSA Reasoner performs various operations on received symbolic vectors that exceed the HDS threshold. These operations depend on the type of vector that is received. For example, in the case of receiving an unbound workflow request vector that matches the service's description vector, the VSA reasoner builds a response vector that is passed to the message transmit buffer for transmission by the delayed response timer. In other cases, the response to a match may be to transmit a clean version of the noisy vector that was received (clean-up service). Additionally, the reasoner can be tasked to ‘peek’ a received workflow vector and to determine if and when the current service may be called in order to pre-provision the service. This task can also include listening to the progress of a particular workflow as flow control is passed among the component services to ensure that the current service has reached its maximum utility if and when it is invoked. The VSA Reasoner also includes an important sub component called the Vector Encoder, which is used to compile symbolic vectors that semantically describe the supported service and its current utility. These vectors can themselves be constructed from other symbolic vectors using the various methods described in sections 5.1 and 6.2.

8.2.5 Delay Response Timer and Local Arbitration (R5)

A major advantage of the VSA approach is the ability to discover and select services using semantic matching. Section 5.2 described how to extend SV encoding beyond simple matches to include measures of real time utility. Service selection therefore involves choosing the correct service with the highest utility or, if the service is not available, suggesting the nearest semantically matching service. For time critical applications that need to be resilient to changes in network connectivity, robustness can be achieved by distributing multiple copies of services throughout the communications network. Further, within the constraints of our target environment—i.e., field operations in very transient, low bandwidth MANETs— it is critical that we do not use unnecessary bandwidth in order to discover an optimal set of services if a sub-optimal set can meet requirements. Using the delayed-response mechanism (R5) largely addresses this by reducing the number of services that respond; however, network latency sometimes results in situations where multiple responses are received. In such situations, the current requester acts as a *local arbitrator* by inspecting the responses and choosing which responder is the winner. It is described in Section 8.3.3, steps 4–8. Note that while the currently active service behaves as the *final arbiter* for services that do send a match response, the process of *local arbitration* is actually a distributed process. Matching services can and do exclude themselves, as described in Section 8.3.3, steps 6 and 7, without ever transmitting their match response.

The purpose of the Delay Response Timer is to ensure that only services with the best symbolic match and hence the highest utility to perform the task will multicast a response message, thereby saving bandwidth, as described in 8.3.2. The use of a time delay to select resources with the highest utility has previously been used successfully to control the connectivity and growth of a dynamic distributed database architecture known as the Gaian Database [111, 112]. How this mechanism operates can be understood from a simple example of a service that is attempting to offer itself as a candidate to be included in a requested workflow. To do this, it needs to determine that it semantically matches the requested service and then be the first matching service to react by transmitting a *Response* vector. On reception of the workflow vector, the service uses the Comparator to determine its degree of match and the corresponding time delay t_{sm}. The service must also have a particular utility to perform the task, which may be based on a number of factors such as available power, the compute platform that it is operating on, connectivity to other resources required for the task, and so on. The service uses its utility to compute a second time delay, t_{ut}, which ranges from zero where there is the highest utility rising to Δt where the utility is low but still adequate to compute the task. If the utility is not sufficient, then t_{ut} is essentially infinity and the Delay Response Timer will not allow the response vector to be multicast. The Delay Response Timer now computes a total delay of $t_d = t_{sm} + t_{ut}$ and stores this with the message in the Message Transmit Buffer.

8.2.6 Message Transmit Buffer

The Message Transmit Buffer is tasked to transmit any messages stored in the message buffer after the corresponding time delay period has elapsed. The proviso is that no other service has transmitted the same message during the time delay period. Therefore, during the time delay period, the Message Transmit Buffer is compared with the Message Listener Buffer, and if there is a match, then the corresponding message is removed from the transmit buffer.

The next section describes a $python2$ implementation of the VSA Platform including a step-by-step description and flow diagram of both active requester and listeners/responders.

8.3 Implementation

The experimental platform was implemented using Python2 for Simpkin et al. [2], and then later versions were implemented in Python3. The Python3 version was also extended in [3] and [4] with tooling to support real world emulations using the CORE/EMANE [113] real-time network emulator. An ubuntu 14.04 virtual machine running under Parallels[1] was used as the base operating system for all of the experiments. The base hardware used was a 16 GB, 2.8 GHz Intel Core i7 laptop. Service agents are started in their own VM; each of which has a separate IP-Address on the simulated wireless mesh network. To create redundancy in the network and ensure competition between services capable of satisfying a particular workflow step, multiple copies of the same service are enabled by instantiating duplicates into separate VMs. Between and during runs, we can move services in and out of the wireless network, taking them in and out of service.

[1]©1999-2020 Parallels International GmbH. All rights reserved.

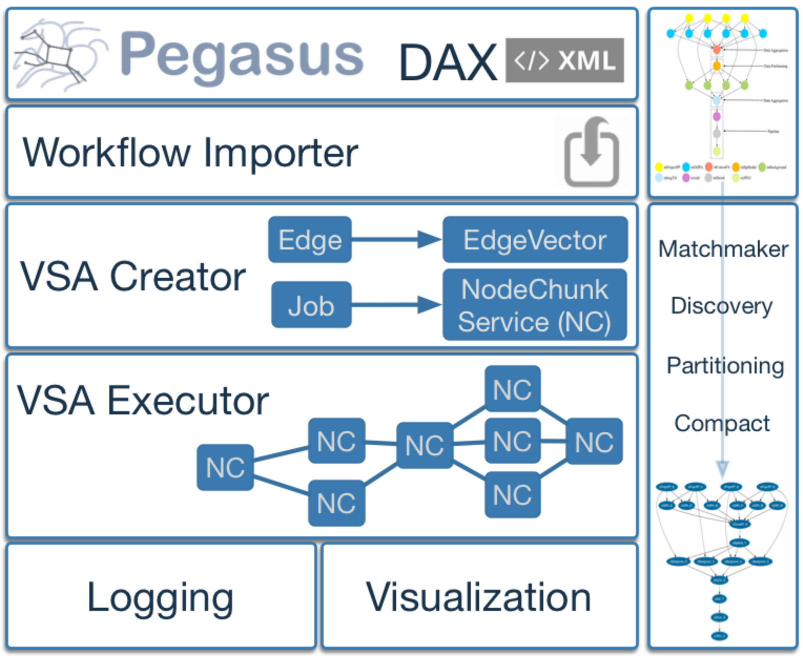


Figure 8.2: Overview of the VSA Implementation platform.

The VSA platform has a modular architecture with several components that are capable of being reused as plugins to other systems. The platform is used to evaluate and demonstrate how symbolic vectors can be automatically constructed from typical scientific workflow representations and how these vectors can then be used to construct, in a decentralized manner, the required workflow in an emulated wireless network environment into which the VSA enabled services are randomly deployed.

- **The Workflow Importer** component can import workflow definitions in various formats including JSON, XML and DAX (a Pegasus workflow definition file [114]). The Workflow Importer consists of a number of bespoke parsers written specifically to convert existing microservice definition files and workflow definition files into service description vectors and workflow request vectors. The workflow request vectors produced are encapsulated in an ordered list: $NodeVectors$, of function+utility description vectors representing the required workflow steps. For DAG workflows, an additional list representing the connections between nodes is produced: the $\boldsymbol{EdgeVectors}$ list. The Workflow Importer passes these lists to the VSA Creator.

- **The VSA Creator** is used to bind the lists of vectors into a single vector, a reduced representation of the workflow using chunking. Chunking is performed bottom up so that higher level vectors are produced as needed. These are recursively bundled until the vector list is reduced to a single vector value. In the case of the static workflow modality, used for discovering and executing DAG workflows, the *NodeVectors* list and *EdgeVectors* list are combined separately producing two *clean-up* Service Vectors (SVs): the $\boldsymbol{Recruit_{Nodes}}$ vector and the $\boldsymbol{Connect_{Nodes}}$ vector. The VSA Creator then bundles these two vectors together with a $\boldsymbol{Start_r}$ role vector into a single vector representing the entire workflow. This *WorkFlow* vector and all its associated sub-vectors are encapsulated in a *chunk tree* object as per Figure 6.1 which is then then passed to the VSA executor.

- **The VSA Executor** is used to instantiate the test case into the CORE/EMANE environment. It *flattens* the workflow by distributing copies of all non-terminal chunk vectors into the terminal (bottom level/worker) nodes. Non-terminal nodes are distributed to the first child of a parent node to ensure that *clean-up services* are actively available. For robustness, the VSA Executor can be made to distribute more than one copy of the clean-up service objects into other terminal node objects.

- **The VSA Layer** consists of the following sub-components:

  – **The VSA Reasoner** is responsible for: *(1)* matching and responding to VSA request vectors as a listener; *(2)* Unbinding and multicasting the next step in a request vector; *(3)* Performing *local arbitration* on responses.

  – **The Comparator** is a Hamming distance function that performs the comparison of vectors to perform the matching.

- **Delay Response Time Engine** is responsible for calculating the fitness function and initiates a delay timer based on the resulting utility.

- **Messaging System** which provides the communications interface for transmitting and listening for vector communications to and from other nodes, along with internal buffers for synchronization with the other system components.
- **The Logging Component** collects metrics as the workflow runs to feed into external processors. Logging currently collects a trace of the nodes and edges that are being processed by the workflow.
- **The Visualisation Component** takes the log output and generates a DAG layout graph using Graphviz [115].

8.3.1 Control Operations

The control of the initiation and subsequent passing of flow control between the different cognitive layers follows a sequence that is the same for all cognitive layers. The **Initiator** performs a subset of the tasks of the cognitive layers to launch and acknowledge the completion of a workflow task by performing the following steps: *(1)* Compile the workflow vector $\boldsymbol{Z_x}$ with a stop vector $\boldsymbol{S_x}$ as the last vector element; *(2)* Transmit $\boldsymbol{Z_x}$; *(3)* Enter collecting mode and listen for response vectors $\boldsymbol{Response_x}$; *(4)* If more than one response, then arbitrate and multicast a *winner* vector, $\boldsymbol{Winner\_Selected}$, to the waiting responders; *(5)* Listen for stop vector $\boldsymbol{S_x}$; and *(6)* On receipt of $\boldsymbol{S_x}$, unbind and transmit to initiate the next workflow step at the same semantic level and then terminate its operation.

The current implementation of the **Cognitive Service Layer** can operate in one of two modes. In the first or *dynamic mode*, the workflow is required to instantiate and run the associated services, on-the-fly, as the workflow unbinding progresses. This mode is applicable to linear workflows. In the second or *static mode* the vector unbinding is used to gather and connect the services into a workflow configuration that is then initiated to perform the required task. This is applicable to more complex workflows with branching and merging requirements. In Chapter 9, examples of both linear and complex workflow use-cases are given.

8.3.2 Service Selection by Local Arbitration

As explained in Section 8.2.5, due to the *Delayed Response* mechanism, *local arbitration* is, in large part, a distributed negotiation. Services '*hear*' responses from better matching services and consequently cancel their own response. However, when more than one response is issued to a request, the currently active node is responsible for arbitrating between responses to select a winner. Using terminology from eq. (6.3) and eq. (6.5), this is achieved in the implementation as follows: if the currently active service is $\boldsymbol{Z'_n}$, then after transmitting the next service request, it enters *match collecting mode* to arbitrate matches from all nodes that reply within a tunable window of time. After the interval expires, the highest ranking responder is selected and a *Winner_Selected* message is broadcast by $\boldsymbol{Z'_n}$ identifying the winner. Since all VSA messages are multicast, all services see all messages, and consequently the winning service continues and losing services discontinue. The $\boldsymbol{Response_v}$ and $\boldsymbol{Winner\_Selected_v}$ vectors are encoded using the currently active workflow vector as a '*tag-id*' as follows:

$$\mathbf{Response_v = Response_r \cdot Z'_n + MyID_r \cdot ID_v + Match_r \cdot Match_v} \tag{8.2}$$

$$\boldsymbol{Winner\_Selected_v = Response_v^1} \tag{8.3}$$

Where

- $\boldsymbol{Response_r}$: is a role vector used to re-encode/permute the received request vector $\boldsymbol{Z'_n}$. This mechanism is a useful way to identify responses since both sender and receiver agree on the original message that was sent.

  - When a requester issues a request, it builds a similar match response vector locally and listens on this '*tag-id*' for replies.
  - When a match is made on the response tag, the requester can use other agreed upon and commonly known role vectors to decode additional information about the responder, including the responder's match quality.

- $\boldsymbol{MyID_r} \cdot \boldsymbol{ID_v}$: allows for differentiating between responders. Each responder generates a unique $\boldsymbol{ID_v}$ and adds this role-filler pair into its $\boldsymbol{Response_v}$ message. After the currently active local arbitrator has examined the responses and transmitted the $\boldsymbol{Winner\_Selected}$ message, each responder can inspect this for its $\boldsymbol{ID_v}$. In this way the winning responder is activated (and the other responders know that they have lost and should not continue).

- $\boldsymbol{Match\_r} \cdot \boldsymbol{Match_v}$: advises the requester of the quality of the responder's match. This is used to choose the best match, but note that the best match will usually be the first responder due to the delayed response mechanism.
  - Responders also use this to inspect other replies while waiting for their own delayed send to expire. If they see a better response, they cancel their response.

- $\boldsymbol{Winner\_Selected_v}$: The winner and losers are easily notified when the requester/local arbitrator simply multicast the winner's $\boldsymbol{Response_v^1}$ vector shifted by one place. The shift is a neat trick that prevents the winner message from looking like a $\boldsymbol{Response_v}$ vector.

8.3.3 Dynamic Workflow Control

In this subsection, we provide a step-by-step account of how a workflow is instantiated as a result of the workflow vector unbinding and matching process. Figure 8.3 shows the logical process for both the Listener and Requester parts of the VSA executor. Each side of the execution logic is described below starting from the listener perspective, assuming that the workflow vector has been unbound and multicast to the network by a previous peer step in the workflow or the workflow requester.

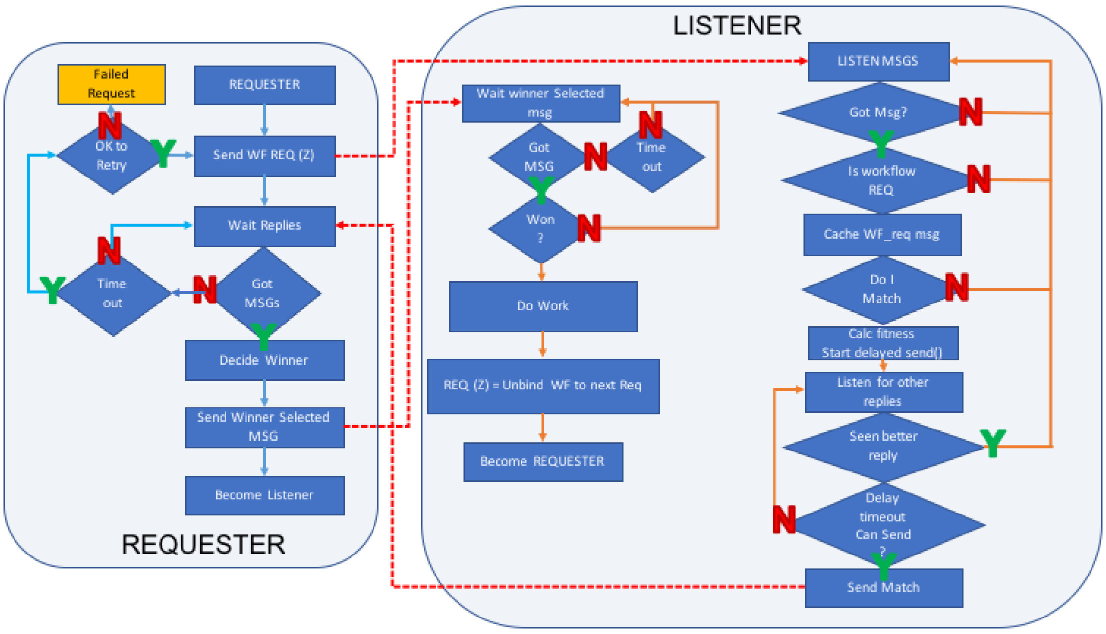


Figure 8.3: Requester / Listener program Flow.

LISTENER:

1. Compile local service description vector Z_s and utility vector Z_u
2. Listen and receive Z'_N
3. Compare service component of Z'_N (i.e. $Z'_N \cdot Serv_r$) with local service vector to compute the semantic match and if there is a match, compute time delay *tsm* based on the Hamming similarity. If no match, return to listening for new vectors.
4. Compare utility component of Z'_N (i.e. $Z'_N \cdot Util_r$) with local utility vector, and if a match, compute time delay *tut* based on the Hamming similarity. If poor utility for the task, return to listening for new vectors.

5. Compute response vector $\boldsymbol{R_N}$.

6. Listen for $\boldsymbol{R_N}$ equivalent vectors from other fitter services for the *delayed-send* period, $\boldsymbol{td = tsm + tut}$. If none are received, then transmit $\boldsymbol{R_N}$.

7. Listen for the $\boldsymbol{Winner\_Selected}$ message. If this node is not the winner, then return to listening for new vectors.

8. If no $\boldsymbol{Winner\_Selected}$ message is received after time-out period, then return to listening for new vectors.

9. On receipt of a $\boldsymbol{Winner\_Selected}$ vector, perform the local service task. We note that this may be a null task or a task to run a sub workflow as a new initiator or simply to perform an action.

REQUESTER:

10. On completion of the local service task, unbind the received vector again to get $\boldsymbol{Z'_{N+1}}$ and transmit.

11. Listen for responses $\boldsymbol{R_{N+1}}$

12. Arbitrate the responses and multicast out the winner message,

    - $\boldsymbol{Winner\_Selected = R^1_{N+1}}$

13. Return to listening for new vectors.

8.3.4 Static Workflow Control

In the static workflow mode, steps 1–12 from the dynamic mode are performed, but rather than terminate at step 13, the cognitive layer computes new semantic vectors that are a combination of the current service name $\boldsymbol{Z_s}$ and its position in the unbound vector to essentially create two temporary service names; a parent-node description vector and a child-node description vector— encoding details of these vectors is given in Section 6.2.4. These vectors are stored in the memory, and the service listens for these vectors. On receipt of a parent or child vector, steps 3–12 are repeated. Step 4 is not required since the name is unique and only this service can respond. In the case of receiving a child vector, the layer also accesses the IP address in the associated message and unicasts a 'hello' message to the associated service to create a connection. The cognitive layer then either listens for new requests, since its service can simultaneously be part of multiple workflows, or waits until the service has completed the current workflow and then resumes listening for new vectors on its original functional service vector description.

8.4 Summary

The VSA Listener/Requester is implemented as a VSA service layer bound to each functional service or sensor. The overall workflow paradigm is that of a one-to-many '*reactive*' discovery architecture. All workflow orchestration messages can be received and acted upon by any listening service. Services compute their compatibility to requests and respond only if they are compatible and do not detect a response from a more compatible service (delayed response mechanism). The comparator is responsible for matching and calculates the delayed response timer interval, which it normalizes relative to the expected match value calculated from the number of sub-vectors embedded in the workflow request. The workflow request vector contains all the information needed to execute the encoded workflow and is *unbound* to expose the next workflow, which is then re-transmitted on the multicast channel by the currently active service. Each unbinding of the workflow vector is a unique permutation of the requested step and is matched upon by listening services. If more than one service responds to a particular workflow request step, then the requesting node arbitrates between responses to chose a winner.

Chapter 9

Use Cases and Evaluation

This chapter describes the experiments taken from my published papers [2–6] that were used to evaluate and verify the thesis hypothesis that:

> A VSA based upon BSCs [48] can be used to define a rich and yet compact encoding that will enable highly efficient representations of multi-modal service descriptions, decentralized service and workflow discovery, and distributed workflow execution. Further, this scheme will provide semantic matchmaking capabilities that can facilitate reasoning on service descriptions and service compositions, or workflows.

9.1 Use Cases

For the evaluation, several use cases were developed and executed on the VSA Platform. Each use case centred on a particular workflow task designed to test aspects of the research claims. Since the VSA Platform brings together all research claims into a single application, note that some of the research claims were addressed by all use cases. In particular, *Scalable hierarchical VSA bundling and semantic matching* (R2), *Bandwidth efficient distributed arbitration* (R4) and *Cognitive workflow model* (R5) together manage flow control/orchestration while minimising the number of messages exchanged during decentralized discovery and winner arbitration. Hence, these aspects were tested by all use cases because they all ran on the VSA Platform.

Therefore, in the list below, entries for research claims (R2), (R4)and (R5)are only included for those use cases where specific evidence was provided.

Below is an overview of the uses cases, highlighting which research claims each supports:

1. **Hamlet use case:**
   This use case models the Shakespeare play Hamlet as a workflow where the words of the play represent the functional worker nodes required to be executed as workflow steps. Higher levels in the play (i.e., acts, scenes, stanzas and sentences) act as '*clean-up*' nodes when the entire play is encoded as a hierarchical workflow as described in Section 6.2.

   (a) **Self-describing, multi-modal semantic service objects** (R1)**:**

   - Each service object maintained two QoS parameters encoded using the method described in Section 5.2. During QoS testing, logs showed that when multiple functional (i.e., word) matches occurred in the network, the service that had the best QoS responded first and was selected as the winner.

   (b) **Scalable hierarchical VSA bundling and semantic matching** (R2)**:**

   - The use case encodes 30k words into sentences, stanzas, scenes, acts and finally a single '*Hamlet*' workflow vector showing that the encoding scheme is scalable and can successfully discover and execute decentralized linear workflows. See Section 9.2.3.
   - During Hamlet playback (i.e. workflow execution), target words were deliberately made unavailable to see that alternate shallow semantic matches (nearest similar word) were found, allowing play back to continue. This demonstrates that the encoding scheme enables semantic matching capability.

   (c) **Holographic dynamic message sizing** (R3)**:**

- The workflow was run with and without dynamic vector truncation to obtain bandwidth saving measures and confirm that the workflow was executed correctly when vector truncation was enabled. On average a 30% bandwidth reduction was achieved for workflow orchestration messages, see Section 9.2.6 and Table 9.6.

2. **Pegasus use case:**
   This use case was designed to validate the DAG encoding scheme described in Section 6.2.4 it makes use of various DAG workflows obtained using the Pegasus workflow generator [114].

   (a) **Self-describing, multi-modal semantic service objects** (R1)**:**

   - Each Pegasus service object encodes its SV description from a Pegasus DAX (XML) file, [109], using the key-value to vector encoding method described in Section 5.4.
   - Each service object maintained two QoS parameters encoded using the method described in Section 5.2. During QoS testing, logs showed that when multiple functional matches occurred in the network, the best QoS service responded first and was selected as the winner.

   (b) **Scalable hierarchical VSA bundling and semantic matching** (R2)**:**

   - A number of Pegasus workflows were used to confirm that DAG workflows can be encoded, discovered and executed using the hierarchical binding scheme in static workflow mode. In this mode, nodes must first be recruited, after which the edge connections are made using dynamically generated parent and child vector addresses, see Section 6.2.4.
   - Scaling was tested using several DAG sizes, up to 1000 nodes, taken from the Pegasus library, see Section 9.2.6, tables 9.2 and 9.3.

   (c) **Holographic dynamic message sizing** (R3)**:**

- This use case was also used to assess dynamic vector truncation by running with and without dynamic vector truncation, see Section 9.2.6
- In addition, the Pegasus *Montage100* workflow was used to assess the effect of chunk size and object similarity. Figure 9.6 shows how compression ratio varies with chunk size and similarity factor and is an experimental verification of the theoretical compression ratios shown in Figure 7.6.

3. **Traffic for London use case:**
   This use case aimed to demonstrate that an existing centralized workflow could be migrated to operate in a decentralized environment by interfacing the existing services with a VSA cognitive adapter/wrapper service. In this way, an existing Node-RED workflow was adapted to run in a decentralized environment emulated using the CORE/EMANE network emulator, see Section 9.1.5. In order to exercise the VSA Platform fully, this use case also incorporated the Pegasus workflow, which could optionally run in parallel to the Node-RED workflow. Operating multiple workflows in parallel verified that the various role vectors employed for encoding SV descriptions (R1)and those used for workflow encoding (R2)were able to maintain separation in the vector space for correct workflow operation.

   (a) **Self-describing, multi-modal semantic service objects** (R1)**:**

   - This use case demonstrated self-describing objects. Each sensor and service object was described by a JSON Node-Red service description file which was converted to a SV description using the JSON to vector method described in Section 5.4. Evidence of semantic comparisons using HDS is can be seen in the workflow operation video and is documented in Section 9.2.5.

   (b) **Scalable hierarchical VSA bundling and semantic matching** (R2)**:**

   - In the video of workflow operation supplied with this use case, many aspects of (R2) can be seen. These are described in sections 9.1.5 and 9.2.5.

(c) **Bandwidth efficient distributed arbitration** (R4)**:**

- The delayed send and local arbitration mechanisms combine to minimize the number of workflow orchestration messages exchanged during workflow execution. This can be seen by inspecting the log screen in the video supplied with this uses case. See also, Section 9.1.5, Figure 9.4 and Section 9.2.5.

4. **Radio reconfiguration:**
This use case was designed to show that autonomous decentralized resources (i.e., battlefield radios) can cooperate, (R5), to complete a task that would otherwise be practically impossible because not all devices in the workflow request are connected to the same network. A secondary objective was to investigate the energy savings that might be achieved by carrying out some vector operations on a neuromorphic Phase-Change Memory device. Thirdly, communication bandwidth savings from dynamic vector truncation, (R3), were assessed. See Section 9.1.7 for full details of this use case.

(a) **Holographic dynamic message sizing** (R3)**:**

- When using dynamic vector truncation, this use case achieved a 2.7x reduction in bandwidth consumption compared to operation without truncation, see Section 9.2.7.

(b) **Cognitive workflow model** (R5)**:**

- This uses case demonstrates the effectiveness of a one-to-many reactive communications paradigm. In the scenario, nodes cache workflow request vectors that they '*hear*' (one-to-many communications) and cooperate to complete the workflow by fulfilling '*takeover requests*' which are issued by the currently active requester when it fails to get a response from the workflow vector's currently active target node (workflow step).

5. **Future Mission Network:**
   This use case shows how VSA can be used to select appropriate services and entities based not only on their functional capabilities but also on dynamically changing variables such as distance from a specified location (R1), see Section 9.1.8.

   (a) **Self-describing, multi-modal semantic service objects** (R1)**:**
   - Each asset creates its SV description from its JSON node description file, see Figure 9.12, using the $json\_to\_vecs()$ algorithm from Section 5.4.
   - In addition, each resource object maintains its position encoded as a 2D *number-line* vector (Section 5.2.1).
   - The task successfully demonstrated resource discovery using vector encoded attribute collections weighted by the resource's position relative to a request position.

9.1.1 Test Data

In order to successfully evaluate and validate the various methods and encodings described in chapters 5, 6 and 7, suitable workflows were required. This proved to be of significant difficulty because while there are many mature, albeit centralized, WFMS available to choose from, there are very few published workflows of sufficient complexity and containing sufficient detail (including service/resource descriptions and workflow specifications) to test the various aspects of my decentralized, scalable VSA approach to workflows.

Using Shakespeare's Hamlet as a large scale linear workflow was originally suggested by Macker and Taylor [116] as a use case for their Network Edge Tool (NEWT).

The play was modeled as a workflow by distributing the actors across a wireless network and having them converse their lines as decentralized messages of communication between one actor and another as the play progressed. I chose to adopt this idea for many of the test cases presented because the hierarchical nature of the play (acts, scenes, sentences, words) mapped perfectly onto my hierarchical binding scheme and it contains many duplicated concepts (words and sentences) so that multiple competing service objects would be available in the network. In addition, there are a total of 29,770 words, of which 4620 are unique. Thus, representing large workflows and sub workflows (acts, scenes, sentences) is possible.

The Pegasus workflow generator [114] was adopted for testing of the static DAG, workflow control, Section 6.2.4, because a wide range of large and small sample DAG workflows are available. Pegasus DAGs are easy to parse because they are specified using an XML description language: the Pegasus DAX file [109]. In addition, images of each sample DAG are available for visual comparison.

9.1.2 Data Collection and Verification

In all of the use cases described below, verification of the outcomes was based on results collected through data logging. In each case, data from the distributed VSA service nodes was captured in one place using a python HTTP log handler. This greatly simplified the aggregation of results for verification against the known test case targets. For example, in the Hamlet use case, the exact order in which the words/sentences are spoken in the play are known. Therefore logging is used to collect the output from the distributed services and a line-check is used to confirm that the play was correctly executed. In the Pegasus use case, the node discovery/recruit phase and edge connection messages were logged from which the output DAG can be created. This same approach, checking of planned/known outcomes, was used for the Radio Reconfiguration and Federated Mission Networks (FMN) use cases. However, in addition to the HTTP logger, the IBM Fyre cloud environment provided additional event logging, network consumption, data logging and graphing, and graphical visualisation of the outcomes.

9.1.3 Dynamic VSA Workflows and QoS - Hamlet workflow

In Simpkin et al. [2](best paper award), to compare with alternative approaches such as those described in [116], the entire text of Shakespeare's play Hamlet was encoded using eq. (6.1) into a hierarchical chunk tree as shown in Figure 9.1. In this example, the component services at the lowest level are the 4620 unique words of the play; the semantic level above are the individual stanzas spoken by each character (not shown in the diagram); the level above this are individual scenes of the play(e.g., A1S1, A1S2); next are the five acts, A1-A5 and then finally a single 10kbit vector semantically represents the whole play (Hamlet). A vector alphabet, a unique vector per alphabet char-

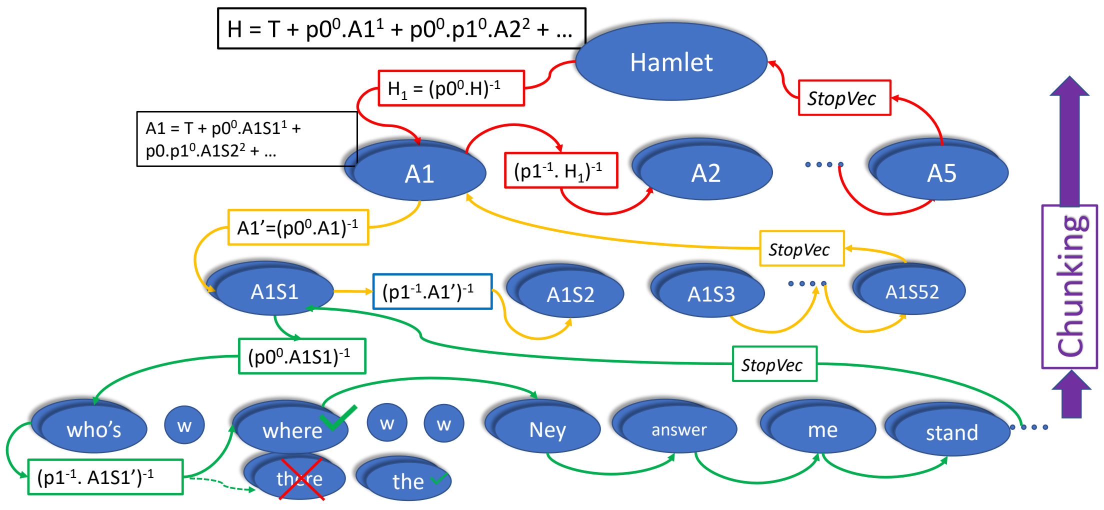


Figure 9.1: Hamlet as a serial workflow

acter, was used to build compound vectors for each word-service in the play. The idea is that each letter making up a word represents some feature of a service description, i.e., analogous to the different input/output/name/descriptions parts of a real world service, as described in Section 5.1. Thus, variable lengths of words and similarity of spellings represent a mix of different services of different complexity and functional compatibility. At the next level, sentences represent a more complex mix of variable length sub-workflows, and so on. It is important to recognise that the higher level vectors do semantically represent the recursive bundling of all the levels below. This fact is used to allow alternative, semantically similar, service compositions to be invoked if the best matching composition is not available. Figure 9.1 shows the word service '*where*' being invoked, in preference to the '*the*' service that also responded, as an alternate to the requested '*there*' service that was unavailable. (This type of syntactic matching is entirely appropriate when modelling service descriptions; deep semantic similarity is unnecessary for this test case.) Note also that when '*where*' completes, it automatically synchronises to the original workflow because it simply *unbinds* the next step from the original workflow vector it received.

Multiple copies of the individual component vectors (words, sentences, acts...) from each level in the hierarchy were distributed in my CORE/EMANE test network and by multicasting the top level vector, the whole play is performed in a distributed manner with 29,770 component word services being invoked in the correct order. Note that the CORE/EMANE network emulator could not support more than about 46 nodes when running on my 16 GB, 2.8 GHz Intel Core i7 laptop. Therefore, only sections of the play could be executed using CORE/EMANE in one run. Larger runs were achieved by using local (OSX) Listener/Requester nodes and limiting the number of duplicate '*word services*' instantiated. The entire play was also encoded/decoded in a single-threaded looping mode that calls the exact same VSA methods I implemented for the service Listener/Requester code (bind,unbind,bundling,matching, etc.).

QoS experiments

In Simpkin et al. [4], the QoS encoding described in Section 5.2 was simulated using two random variables: *current_load* and *battery_life*. From the requester's point of view, responders having minimum *current_load* and maximum *battery_life* should respond first. Acceptable ranges of values were randomly chosen when encoding service request vectors, and each service simulated its own QoS in the same manner. Thus, matching on functional as well as QoS criterion was tested. Verification of correct matching for the QoS experiments was done by noting that, when multiple responses were received, the best responder should have logged the minimum *current_load* value and the maximum *battery_life* value. The VSA Listener code was updated to output these values in the '***PWORK***' and '***WORK***' logger entries (see Figure 9.4 for an example of multiple responders).

9.1.4 Static VSA Workflows - Pegasus DAG workflows

Simpkin et al. [2] and [4] also presented an evaluation of Section 6.2.4 which describes how more complex DAG workflows can be discovered and connected using the VSA hierarchical binding scheme (R2). The main objectives of this experiment were to;

- demonstrate that DAG workflows can be encoded into a symbolic vector representation and then recursively decoded to assemble the required workflow in a decentralized setting,
- show that the workflow constructed was also resilient to changes in the communications network, and
- to demonstrate that semantic matching on SV descriptions (R1), built from Pegasus DAX workflow description files, can be used to discover and recruit service nodes into the DAG without knowledge of their IP-Address locations.

For the evaluation, we used five different DAG workflows, generated using the Pegasus workflow generator [114]:

1. Montage (NASA/IPAC) stitches multiple input images together to create custom mosaics of the sky.

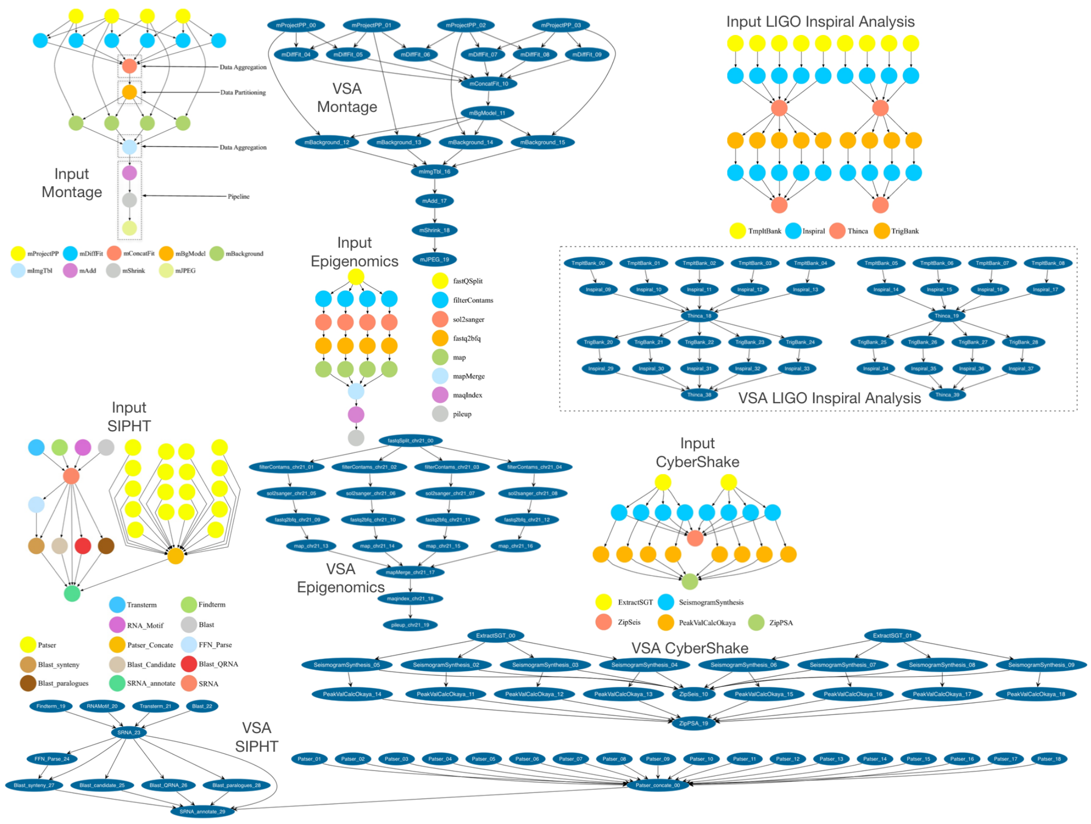


Figure 9.2: A comparison of five different DAX workflows as input and the VSA reconstructed workflows from post processing the semantic vector.

2. CyberShake (Southern Calfornia Earthquake Center) characterizes earthquake hazards in a region.

3. Epigenomics (USC Epigenome Center and Pegasus) automates various operations in genome sequence processing.

4. Inspiral Analysis (LIGO) generates and analyzes gravitational waveforms from data collected during the coalescing binary systems.

5. SIPHT (Harvard) automates the search for untranslated RNAs (sRNAs) for bacterial replicons in the NCBI database.

We again ran a series of experiments using the CORE/EMANE network emulator to simulate a MANET network. Pegasus DAX workflows were processed using the VSA creator to build the semantic vector workflow encodings and also to generate the service description vectors that semantically describe each of the component services. Each service creates its SV description from the Pegasus DAX job step entries to which it is assigned and listens for multicast work requests that match '*self*'. Note that many of the underlying node process specified in these workflows are highly compute intensive so that they cannot be run in a laptop environment, meaning that once the DAG was formed and connected, stub services were used in place of the real Pegasus modules. The workflow request vector was launched from some node in the network, and the workflow was constructed in a decentralized manner, with control being passed between services as the workflow vector was recursively unbound. During the workflow execution process a range of metrics was extracted which provided a detailed log of the run and the order of execution. The nodes and edges selected during the run was extracted from the log, and a visual result was displayed using Graphviz. Figure 9.2 shows the results for the five different Pegasus workflows evaluated. The coloured images represent the Pegasus generated workflows and blue workflows show the VSA generated reconstruction of the workflows. Aside from the cosmetic difference, this demonstrates that all workflows were composed and correctly connected in all cases.

To demonstrate the resilience of the approach, the network connectivity was modified by moving nodes in and out of operation in order to demonstrate that different instances of the correct services were selected and that this still produced the same required workflow. However, since DAG workflows are discovered and connected before they are started, see Sections 6.2.4 and 8.3.4, note that I did not attempt to implement recovery methods for the case when a node fails during DAG workflow operation (an area requiring future work).

For the QoS investigations, the same testing and verification approach described in Section 9.1.3 was used, and it showed that when multiple similar services respond to a request, the service with the highest utility was selected in preference to services having lower utility.

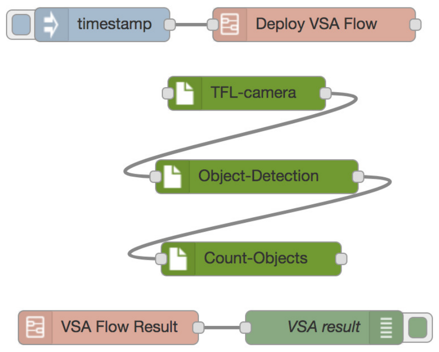
9.1.5 Node-RED Integration - Traffic Congestion

The main focus of Simpkin et al. [3] was to investigate means whereby Node-RED workflows can be migrated to operate in a decentralized execution environment, so that such workflows can run on Edge networks, where nodes are extremely transient in nature. The objectives were to:

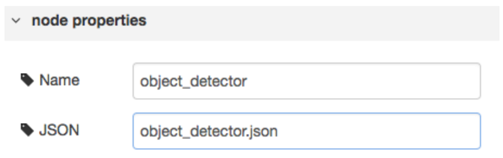
1. Confirm that JSON service descriptions and service request steps can be vectorized, (R1), and that these vector descriptions can be used to discover and execute decentralized workflows (R2), (R4) and (R5).
2. Highlight the fact that matching on vectorized JSON descriptions is greatly simplified because all that is needed is to perform a hamming distance or dot product match.
3. Show that the VSA approach described in this thesis can be used to extend a centralized architecture and allow it to discover and execute decentralized workflows without the need to specify IP-Address locations and without having all messages pass through the central controller.

The work demonstrated the feasibility of such an approach by showing how an existing Node-RED based traffic congestion detection workflow could be migrated into a decentralized environment.

In the scenario, an existing Node-RED traffic congestion use case, Harborne et al. [117, 118] was simplified to count the number of cars on a given street by discovering a sensor service in the correct location, connecting this to an available *object_detector* having the appropriate input/output data types and classification abilities, and finally connecting the object detector to a '*counting_service*', see Figure 9.3. The sensor, object detector, and counting services were implemented as RESTful *flask* [119] wrapper services, termed Workflow-Agents (WAs) in the paper. The WAs act as shims to the



(a) VSA workflow composition using vsa_service node.



Name: object_detector
JSON: object_detector.json

(b) Node-RED vsa_service properties.

Figure 9.3: VSA enabled Node-RED workflow.

existing Node-RED local or remote services and use a defined set of endpoints to communicate with the VSA cognitive layer and also to send results and receive workflow requests to and from Node-RED.

The Transport for London (TFL) traffic camera network and API[1] were used for the sensor services. These allows access to imagery and video from around one-thousand traffic cameras situated around London. The imagery and video are updated every five minutes, and the video provided is a ten-second clip recorded at the beginning of the five minute interval. The location and capabilities (e.g., resolution and latency) of each camera were also specified using JSON service description files.

Multiple (including duplicate) TFL camera, object detector and counting services, having differing capabilities described using JSON similar to that shown in Listing 5.3, were instantiated in the CORE network. Each WA activates its VSA workflow layer by passing its JSON description file to the VSA importer. The importer converts this to a VSA semantic vector using the algorithm listed in Section 5.5 and originally described in [3]. In effect, this converts the existing Node-RED microservices into a cooperating set of decentralized proxies, which are instantiated into the CORE environment by

[1]http://www.trafficdelays.co.uk/london-traffic-cameras/

adding a cognitively-aware wrapper around each service to facilitate decentralized discovery and execution. Workflows are then composed using the Node-RED graphical interface to describe the connectivity and functional requirements of each workflow step but without specifying IP locations as per the normal Node-RED scheme. The user interface is then used to launch the workflow request into the CORE environment for distributed discovery and execution, and the results are returned to Node-RED for display. In addition to the traffic congestion workflow, the new Node-RED mode was used to graphically create and run a distributed simulation of the Montage Pegasus[114] workflow in the CORE/EMANE environment.

For further details of the scenario and use case details see Simpkin et al. [3].

Results

A video of the Node-RED use case, which is narrated below with time-stamps, is available at:

`https://drive.google.com/file/d/1Bore6cRnY19R0rfiaM-EngUYUBrVhphG/view?usp=sharing`

To understand the video, an explanation of some of the log entry terms is given first:

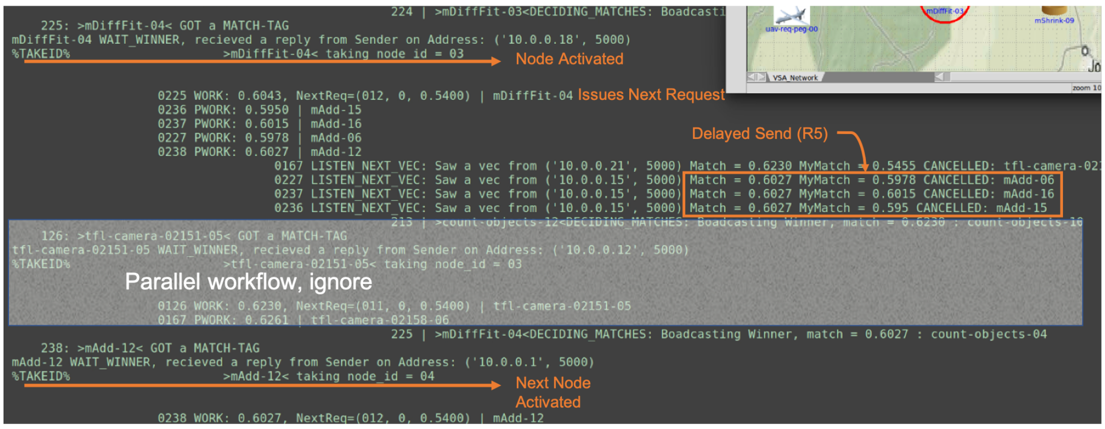
- **PMATCH: 0.6427 | node_name:** This indicates that a clean-up node has matched, with the HDS value shown, and is a potential (P) candidate to execute its sub-workflow.
- **MATCH: 0.6302 | node_name:** Indicates that the node knows it is selected as the winner and is carrying out its clean-up service (by descending the workflow hierarchy, Section 6.2.2).
- **PWORK: 0.6427 | node_name:** This indicates that terminal/worker node has matched, with the HDS value shown, and is a potential (P) candidate to do the work.

- **WORK: 0.6136 | node_name:** Indicates that the terminal/worker node knows it is selected as the winner and is carrying out its function, after which it will unbind the workflow vector to traverse horizontally in the workflow hierarchy, Section 6.2.2).

- **%TAKEID%** shows a node being selected during the recruitment phase. It calculates its position in the workflow, i.e., $NodeID$, which is used to calculate its parent and child vector IDs. These vector IDs are used during the $Connect_{Nodes}$ phase to connect the DAG, section 6.2.4.

- **%REQCON%** shows a node acting as a parent (producer) node during the $Connect_{Nodes}$ phase requesting a connection to a child (consumer) node. The IP-Address listed is the node's own address (can be cross referenced in the log when the workflow is started).

- **%TAKCON%** shows a node acting as a child node accepting a request for connection. The IP-Address listed is the node's own address.

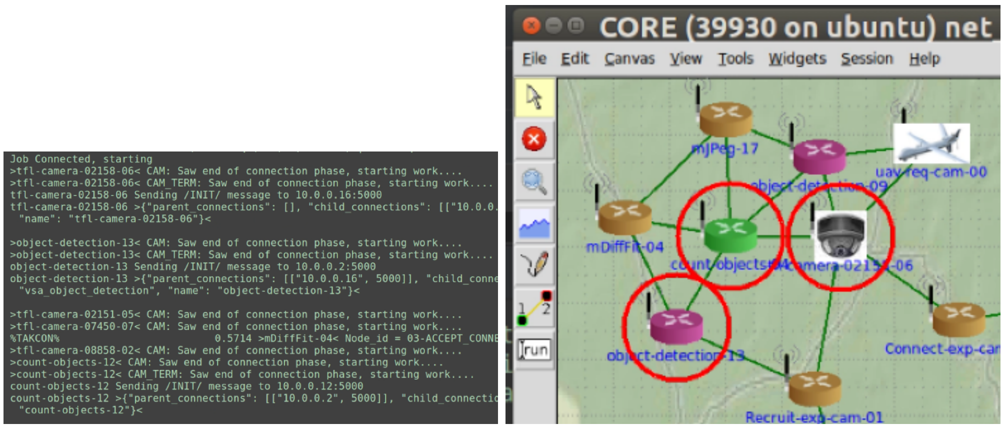


```
                                                   224 | >mDiffFit-03<DECIDING_MATCHES: Boadcasti
    225: >mDiffFit-04< GOT a MATCH-TAG
mDiffFit-04 WAIT_WINNER, recieved a reply from Sender on Address: ('10.0.0.18', 5000)
%TAKEID%                     >mDiffFit-04< taking node id = 03

                    0225 WORK: 0.6043, NextReq=(012, 0, 0.5400) | mDiffFit-04
                    0236 PWORK: 0.5950 | mAdd-15
                    0237 PWORK: 0.6015 | mAdd-16
                    0227 PWORK: 0.5978 | mAdd-06
                    0238 PWORK: 0.6027 | mAdd-12
                                   0167 LISTEN_NEXT_VEC: Saw a vec from ('10.0.0.21', 5000) Match = 0.6230 MyMatch = 0.5455 CANCELLED: tfl-camera-02
                                   0227 LISTEN_NEXT_VEC: Saw a vec from ('10.0.0.15', 5000) Match = 0.6027 MyMatch = 0.5978 CANCELLED: mAdd-06
                                   0237 LISTEN_NEXT_VEC: Saw a vec from ('10.0.0.15', 5000) Match = 0.6027 MyMatch = 0.6015 CANCELLED: mAdd-16
                                   0236 LISTEN_NEXT_VEC: Saw a vec from ('10.0.0.15', 5000) Match = 0.6027 MyMatch = 0.595 CANCELLED: mAdd-15
                                                   213 | >count-objects-12<DECIDING_MATCHES: Boadcasting Winner, match = 0.6230 : count-objects-10
    126: >tfl-camera-02151-05< GOT a MATCH-TAG
tfl-camera-02151-05 WAIT_WINNER, recieved a reply from Sender on Address: ('10.0.0.12', 5000)
%TAKEID%                     >tfl-camera-02151-05< taking node_id = 03

                    0126 WORK: 0.6230, NextReq=(011, 0, 0.5400) | tfl-camera-02151-05
                    0167 PWORK: 0.6261 | tfl-camera-02158-06
                                                   225 | >mDiffFit-04<DECIDING_MATCHES: Boadcasting Winner, match = 0.6027 : count-objects-04
    238: >mAdd-12< GOT a MATCH-TAG
mAdd-12 WAIT_WINNER, recieved a reply from Sender on Address: ('10.0.0.1', 5000)
%TAKEID%                     >mAdd-12< taking node_id = 04

                    0238 WORK: 0.6027, NextReq=(012, 0, 0.5400) | mAdd-12
```

Figure 9.4: Delayed Send and Local Arbitration

Video narration:

- $00:06$ shows two workflows being activated using the Node-RED interface, the traffic congestion workflow and a Pegasus montage workflow, which will be



```
Job Connected, starting
>tfl-camera-02158-06< CAM: Saw end of connection phase, starting work....
>tfl-camera-02158-06< CAM_TERM: Saw end of connection phase, starting work....
tfl-camera-02158-06 Sending /INIT/ message to 10.0.0.16:5000
tfl-camera-02158-06 >{"parent_connections": [], "child_connections": [["10.0.0.
  "name": "tfl-camera-02158-06"}<

>object-detection-13< CAM: Saw end of connection phase, starting work....
>object-detection-13< CAM_TERM: Saw end of connection phase, starting work....
object-detection-13 Sending /INIT/ message to 10.0.0.2:5000
object-detection-13 >{"parent_connections": [["10.0.0.16", 5000]], "child_conne
  "vsa_object_detection", "name": "object-detection-13"}<

>tfl-camera-02151-05< CAM: Saw end of connection phase, starting work....
>tfl-camera-07450-07< CAM: Saw end of connection phase, starting work....
%TAKCON%                         0.5714 >mDiffFit-04< Node_id = 03-ACCEPT_CONNE
>tfl-camera-08858-02< CAM: Saw end of connection phase, starting work....
>count-objects-12< CAM: Saw end of connection phase, starting work....
>count-objects-12< CAM_TERM: Saw end of connection phase, starting work....
count-objects-12 Sending /INIT/ message to 10.0.0.12:5000
count-objects-12 >{"parent_connections": [["10.0.0.2", 5000]], "child_connectio
  "count-objects-12"}<
```

Figure 9.5: TFL Camera, traffic workflow starting.

executed in parallel. (Note that the image will have changed by the end of the demonstration because the workflow will have retrieved a live image for processing.)

- **00 : 10** The CORE/EMANE graphical interface displays a mesh network with various nodes positioned haphazardly.
- **01 : 28** shows the individual nodes being started in CORE/EMANE, each having a different IP-Address (which is unknown to the VSA workflow).
- **02 : 22** shows the recruit phase of both the traffic (Recruit_exp_cam) and Pegasus workflows (Recruit_exp_peg) sending their respective $\boldsymbol{Recruit_{nodes}}$ vectors labeled as “sending start command”. The respective clean-up services instantiated on individual nodes announce that they have detected a $\boldsymbol{PMATCH}$. In the case of the Pegasus workflow, the HDS match quality indicates 0.6427, and for the traffic workflow, the match quality is 0.6302.
- **03 : 04** Shows that the delayed send, Section 8.3.2, causes nodes to actively cancel their planned transmission on seeing a better match, Figure 9.4.
- **04 : 05** Shows the discovered traffic workflow connected and operating Peer-to-Peer (P2P), Figure 9.5.

1. The “Job Connected, Starting” notification is seen.
2. The “Sending /INIT/ message to ...” entries show the flask services belonging to the VSA layer being started.
3. The line following (2) shows how each discovered node is informed of the IP-Addresses of each of its parent and child partners (as well as the return route to Node-RED or keyword ‘DMY’ if none is appropriate) using a JSON POST message.

- **05:05** Shows the discovered Pegasus montage workflow connected and operating Peer-to-Peer (P2P). The “Job Connected,|Starting” notification is seen. The “Sending /INIT/ message to ...” entries show the flask services belonging to the VSA layer being started
- **05 : 33** shows the Node-RED composition of the target Pegasus workflow and that it correctly terminated on an mJeg object (mJpeg-10). Again, the selected node names, flask IP-Addresses and so on can be inspected in this part of the log.
- **05:37** shows the that a new camera image was retrieved and that the number of cars detected was thirteen.

9.1.6 Binary Spatter Code Message Truncation (R5)

Chapter 7 and Simpkin et al. [5] describe the mathematical model for BSC vector truncation (R4). This section reports on the empirical results obtained in [5]. The truncation model was applied to the Hamlet and Pegasus use cases described in sections 9.2.3 and 9.2.4. The objective was to investigate how much bandwidth could be saved by truncating the vector requests for both workflow discovery and orchestration based on the number of sub-vectors that each VSA concept vector contains and for various levels of similarity between Service Vector (SV) descriptions in the network. The VSA Platform was updated to perform vector truncation on the fly at the point of multicast transmission. All SV descriptions and workflow graphs were built in the usual way using 10kbit VSA vectors. The discovery and orchestration scheme remained unchanged as described in sections 6.2 and 8.3.3, except for the inclusion of the minimum vector size calculation and truncation of the vectors before transmission.

The owner of every VSA concept vector knows how many sub-vectors it contains (since it was built by adding sub-vectors) and therefore it can use the information contained in Figure 7.6 to calculate the minimum vector size needed to transmit its vector. For example, the 'Hamlet' workflow vector contains seven sub-vectors (the five acts plus some meta-data). If the system can assume it is safe to use a similarity factor of 50% then Figure 7.6 indicates that the 10kbit vector can be truncated to 1680 bits. The vector is unbound to reveal the Act_1 vector then truncated to 1680 bits before it is multicast to the network. Each service compares this vector with the first 1680 bits of their vector and measures the Hamming distance. Since the number of bits is now 1680, the service computes the threshold from the information in Figure 7.7, which is a value of 0.42, and if the measured hamming distance is less than the threshold then the service follows the protocol for determining if it should respond or not. Should the service calculate itself to be a good match, then before responding, it must truncate its reply vector to the minimum understandable length. To do this, it counts the number of sub-vectors contained in its response, which is 10; therefore, from the model, it will truncate its vector to 2751 bits before transmission.

Table 9.1 shows the results obtained, via logging, for the bandwidth savings achieved using the Hamlet linear workflow test-case. For this test-case we used two different word binding methods, positional and XOR chaining, when building the representation so that we were able to manufacture different levels of similarity. The positional binding scheme creates words vectors that are similar to each other whereas the XOR chaining scheme creates unique vectors for words (but not sentences, since the positional scheme was used at the sentence level in both cases). In Table 9.1, column *'10k'* is the bandwidth consumed without vector truncation. Columns *'s60'* and *'s50'* are the

Workflow	10k	s60	s50	s60%	s50%
ACT_1	72.71	57.15	49.57	78.60	68.18
ACT_2	61.74	48.74	43.14	79.37	69.87
ACT_3	78.25	62.21	54.39	79.50	69.52
ACT_4	55.55	45.25	38.72	81.46	69.71
ACT_5	59.37	46.93	40.86	79.06	68.83

Table 9.1: Bandwidth savings (MB) and compression ratio for Hamlet workflow, chunk size variable based on sentence length.

Workflow	10k	s50	s40	s50%	s40%
Epigen_24	0.87	0.45	0.35	53.23	41.94
Montage_25	1.24	0.66	0.52	56.51	44.15
Inspiral_40	1.43	0.77	0.60	58.08	45.54
Inspiral_100	3.66	2.03	1.59	53.85	41.96
Montage_100	6.07	3.43	2.68	55.46	43.44
Epigen_997	34.48	19.95	15.62	58.09	45.35
Inspiral_1k	33.69	19.57	15.28	51.72	40.23
Montage_1k	59.07	34.31	26.90	57.86	45.30

Table 9.2: Bandwidth savings (MB) and compression ratio for discovery of various Pegasus worklfows, $ChunkSize = 23$.

bandwidths consumed when a similarity factor of *s60=60%* (used for the positional binding scheme) and *s50=50%* (used for the XOR binding scheme). Columns *'s60%'* and *'s50%'* are the compression ratios obtained for the respective similarity factor. It is interesting to note that we could not get consistently clean runs when using a similarity factor of *'s50%'* and the positional binding scheme, nevertheless the positional binding scheme allows for better semantic matching of the sentence and word concepts we are using to model workflow and service objects in this test-case.

Workflow	10k	s50	s40	s50%	s40%
Epigen_24	0.83	0.51	0.40	60.94	48.19
Montage_25	1.22	0.79	0.62	64.96	50.82
Inspiral_40	1.39	0.90	0.69	64.32	49.64
Inspiral_100	3.57	2.49	1.92	67.71	53.78
Montage_100	5.89	4.16	3.24	70.56	55.01
Epigen_997	32.54	23.55	18.69	72.39	57.44
Inspiral_1k	31.97	23.13	18.18	72.36	56.87
Montage_1k	57.07	41.51	32.44	72.74	56.84

Table 9.3: Bandwidth savings (MB) and compression ratio for discovery of various Pegasus worklfows, $ChunkSize = 29$.

Tables 9.2 and 9.3 show the bandwidth savings obtained using the same approach for discovery and connection of various Pegasus workflow examples. There is less similarity between differing service objects in these Pegasus examples, so these runs were conducted at similarity factors of *'s50'* and *'s40'*. The savings listed represent purely the savings in the message bandwidth generated for discovery and orchestration; no actual data processing was executed. the Pegasus workflow examples were used as a means to test that the VSA Platform could successfully encode, discover and connect DAG workflows in a verifiable way when employing truncation. Verification of correct runs was carried out as described in Section 9.2.4.

Figure 9.6 shows how compression ratio varies with chunk size and similarity factor and is an experimental verification of the theoretical compression ratios shown in Figure 7.6. All graphs where obtained by building the Montage_100 test case at various chunk sizes and then running discovery on these workflows using different similarity factors. The obvious conclusion from Figure 9.6 is that small chunk sizes give better compression ratios and it should be noted that the recursive nature of eq. (6.1) enables the use of small chunk sizes for concepts containing many sub-features. However, the

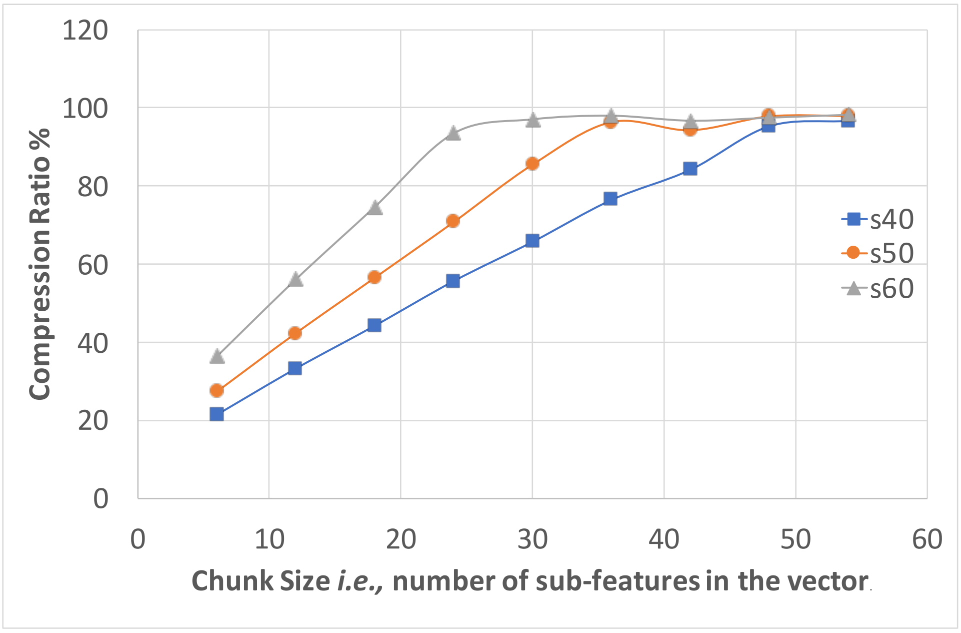


Figure 9.6: Minimum message size (No Bits) for chunk vectors containing n sub-feature vectors.

results in section 6.3 and originally shown in [4] suggest that smaller chunk sizes can reduce the semantic matching capabilities of the resulting concept vectors. This is an area for future investigation.

9.1.7 Radio Reconfiguration with Neuromorphic Phase-change Memory Integration

This use case was originally described in Bent et al. [6] and represents a strong validation of the research presented in this thesis because it is one of only a few separately funded technology transition contracts to be awarded by my sponsor, the U.K. Ministry of Defence (MOD).

The scenario is loosely based around the Anglova tactical military scenario [120] and

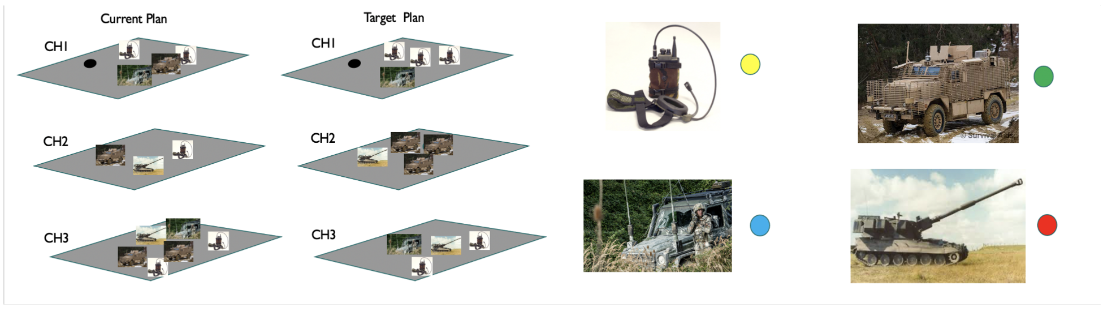


Figure 9.7: Schematic of the scenario showing the current and target communication plans in three channel layers. The colour coding of the different radio/platforms is used to illustrate the vector unbinding .

experimentation environment developed by the NATO IST-124 Research Task Group. In the scenario, different types of platforms and their associated radios, operating in a low bandwidth Tactical Communications and Information Systems (TacCIS) environment, are assigned to the one of three different UHF radio channels, as shown in Figure 9.7. Some of the radios are assumed to be on a fixed channel; some can operate on any two channels, and a minority can operate on all three channels. After some events in the field, a commander requires a number of the distributed assets and services to switch from their current radio channel assignments to a new set of channel assignments. The approach used is particularly applicable to a future military Internet of Battlefield Things (IoBT) where rapid reconfiguration of assets is required to support changing mission needs. The main objectives were:

1. to demonstrate that a set of VSA-enabled resources can cooperate in a decentralized P2P manner (R5)using the hierarchical VSA workflow scheme (R2) to carry out a reconfiguration task without central control and without nodes having knowledge of IP-locations and even in the case when the target node does not share a common channel with the current requester.

2. to investigate the bandwidth savings obtained when vector truncation (R3) is used as described in Section 7.2.

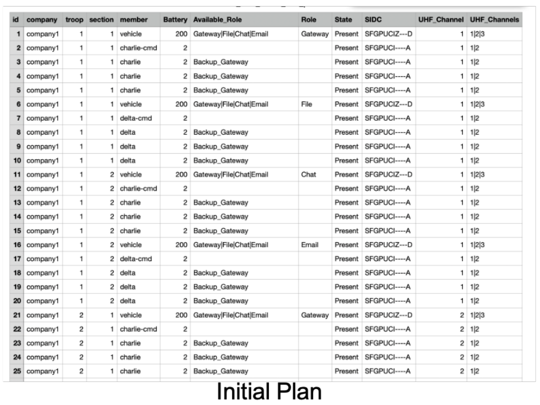
3. to show how a neuromorphic phase-change memory (PCM) chip might be incorporated into the VSA workflow architecture and assess the potential energy benefit of using such a device.

My contribution to this paper was to extend the implementation of the VSA Platform to meet objectives 1) and 2) and show how the reactive one-to-many communication paradigm (R5) facilitates cooperation between autonomous nodes to complete such a difficult decentralized configuration task. The main extension was the introduction of the '*take-over*' request message, which is used when the current workflow requester node fails to get a response to its unbinding and multicast of the workflow request vector. The purpose is to find a peer node that will act as a proxy for the requester so that the request vector can be multicast on channels to which it does not have access. The *take-over* request specifies the channels on which the workflow request has been made so that only listening nodes that have access to channels that have not been visited will respond. The VSA Platform used its caching of previously seen workflow request vectors to easily implement this behaviour. For more details of how the Platform was extended to cater in such situations, see [6, Sections 5, 6, 7].

The Anglova environment and radio modeling was created by the IBM Hursley authors. The PCM investigations were carried out by the IBM Zurich authors and will not be included here because I did not contribute to this section of the paper (except by supplying access to BSC vectors for use in the PCM experiment). For more information on the PCM part of the paper see [6, Section 4, page 6-9]. The implications of the PCM experiment and findings with respect to this thesis are reserved for Section 9.2.7.

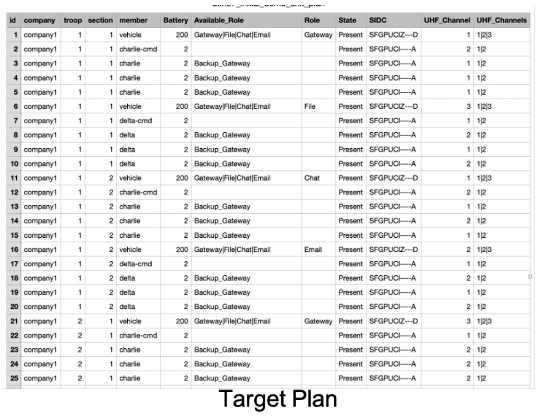
Experimental Environment and Data Collection

The simulation, built by the IBM Hursley authors, was performed in the IBM Fyre cloud environment using a Ubuntu 20.04 VM, which has been set up with the Anglova network emulation scripts, a Mosquitto MQTT server and the Node-RED orchestration



id	company	troop	section	member	Battery	Available_Role	Role	State	SIDC	UHF_Channel	UHF_Channels
1	company1	1	1	vehicle	200	Gateway|File|Chat|Email	Gateway	Present	SFGPUCIZ---D	1	1|2|3
2	company1	1	1	charlie-cmd	2			Present	SFGPUCI----A	1	1|2
3	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
4	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
5	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
6	company1	1	1	vehicle	200	Gateway|File|Chat|Email	File	Present	SFGPUCIZ---D	1	1|2|3
7	company1	1	1	delta-cmd	2			Present	SFGPUCI----A	1	1|2
8	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
9	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
10	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
11	company1	1	2	vehicle	200	Gateway|File|Chat|Email	Chat	Present	SFGPUCIZ---D	1	1|2|3
12	company1	1	2	charlie-cmd	2			Present	SFGPUCI----A	1	1|2
13	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
14	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
15	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
16	company1	1	2	vehicle	200	Gateway|File|Chat|Email	Email	Present	SFGPUCIZ---D	1	1|2|3
17	company1	1	2	delta-cmd	2			Present	SFGPUCI----A	1	1|2
18	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
19	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
20	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
21	company1	2	1	vehicle	200	Gateway|File|Chat|Email	Gateway	Present	SFGPUCIZ---D	2	1|2|3
22	company1	2	1	charlie-cmd	2			Present	SFGPUCI----A	2	1|2
23	company1	2	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
24	company1	2	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
25	company1	2	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2

Initial Plan



id	company	troop	section	member	Battery	Available_Role	Role	State	SIDC	UHF_Channel	UHF_Channels
1	company1	1	1	vehicle	200	Gateway|File|Chat|Email	Gateway	Present	SFGPUCIZ---D	1	1|2|3
2	company1	1	1	charlie-cmd	2			Present	SFGPUCI----A	2	1|2
3	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
4	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
5	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
6	company1	1	1	vehicle	200	Gateway|File|Chat|Email	File	Present	SFGPUCIZ---D	3	1|2|3
7	company1	1	1	delta-cmd	2			Present	SFGPUCI----A	1	1|2
8	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
9	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
10	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
11	company1	1	2	vehicle	200	Gateway|File|Chat|Email	Chat	Present	SFGPUCIZ---D	1	1|2|3
12	company1	1	2	charlie-cmd	2			Present	SFGPUCI----A	2	1|2
13	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
14	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
15	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
16	company1	1	2	vehicle	200	Gateway|File|Chat|Email	Email	Present	SFGPUCIZ---D	2	1|2|3
17	company1	1	2	delta-cmd	2			Present	SFGPUCI----A	1	1|2
18	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
19	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
20	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
21	company1	2	1	vehicle	200	Gateway|File|Chat|Email	Gateway	Present	SFGPUCIZ---D	3	1|2|3
22	company1	2	1	charlie-cmd	2			Present	SFGPUCI----A	1	1|2
23	company1	2	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
24	company1	2	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2
25	company1	2	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	2	1|2

Target Plan

Figure 9.8: Initial and target communications plans showing the first 25 radio definitions.

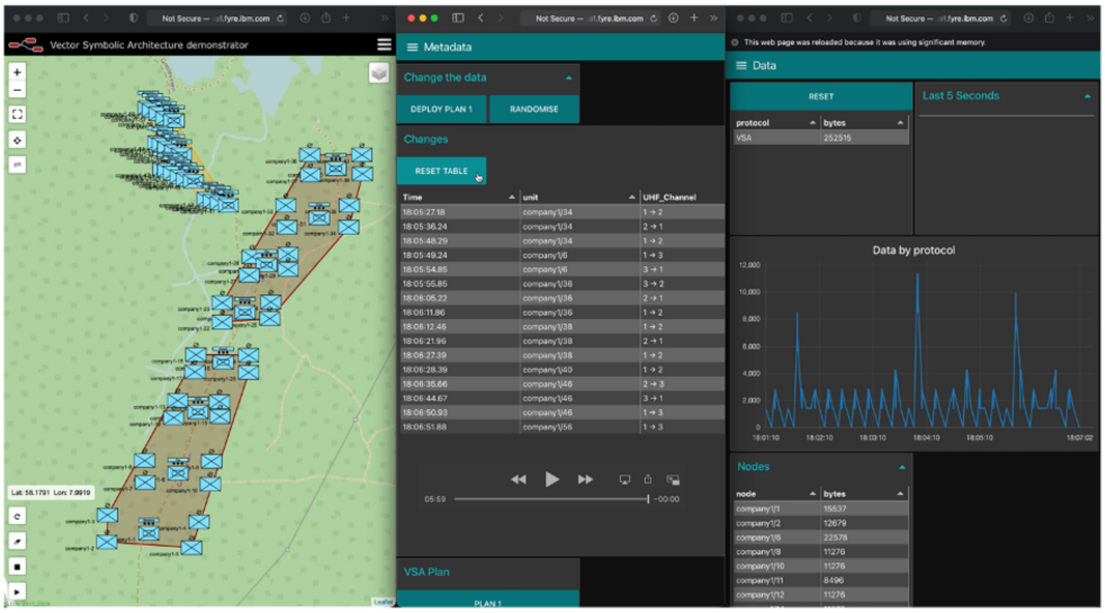
engine and network packet capture. Each radio, its associated platform and the code required to perform the VSA operations is encapsulated in Linux Containers running EMANE. For more information see [6, Section 8.1, page 16].

Experimental Results

The demonstration scenario comprises sixty radio assets, where each asset is described by a set of characteristics and is initially assigned to a specific radio channel. Each asset is also given a list of the channels on which it is accredited to operate. The initial and target communications plans are shown in Figure 9.8 for the first 25 radios in the scenario.

This target plan is requesting 20 specified assets to be on Channel 1, 20 on Channel 2 and 20 on Channel 3. For the purposes of the demonstration, twenty out of the sixty assets have been randomly initialized onto channels different from the required designation in the new workflow plan. The objective is to discover these assets and instruct them to move to the required channel. Correct operation can then easily be verified using the environment's graphical display as well as from the logs.

A video of the demonstration is available at:



Time	unit	UHF_Channel
18:05:27.18	company1/34	1 → 2
18:05:36.24	company1/34	2 → 1
18:05:48.29	company1/34	1 → 2
18:05:49.24	company1/6	1 → 3
18:05:54.85	company1/6	3 → 1
18:05:55.85	company1/36	3 → 2
18:06:05.22	company1/36	2 → 1
18:06:11.86	company1/36	1 → 2
18:06:12.46	company1/38	1 → 2
18:06:21.96	company1/38	2 → 1
18:06:27.39	company1/38	1 → 2
18:06:28.39	company1/40	1 → 2
18:06:35.66	company1/46	2 → 3
18:06:44.67	company1/46	3 → 1
18:06:50.93	company1/46	1 → 3
18:06:51.88	company1/56	1 → 3

protocol	bytes
VSA	252515

node	bytes
company1/1	15537
company1/2	12679
company1/6	22578
company1/8	11276
company1/10	11276
company1/11	8496
company1/12	11276

Figure 9.9: The demonstrator screen showing the location of the different radios and the communications that are occurring as the scenario plays out.

`https://drive.google.com/file/d/1HGcVuVRqVJpv8yfKVXmENTzisXyCBQoW/view?usp=sharing`

The layout of the demonstration display is shown in Figure 9.9. On the left hand side is a map that displays the location of the different assets. In the centre of the display are various panels that indicate the actions that are taking place and on the right hand side of the display are panels that show the data transfers (bytes transmitted) as the workflow progresses.

In Figure 9.10, we show the maps corresponding to the location and initial and final configuration of the assets. The shaded polygons, each corresponding to a radio channel, indicate the area within which assets on a particular radio channel are located.

Sequence of Operations

To understand the sequence of operations in the demonstration video, we describe here the first few actions occurring in the scenario to illustrate the different actions

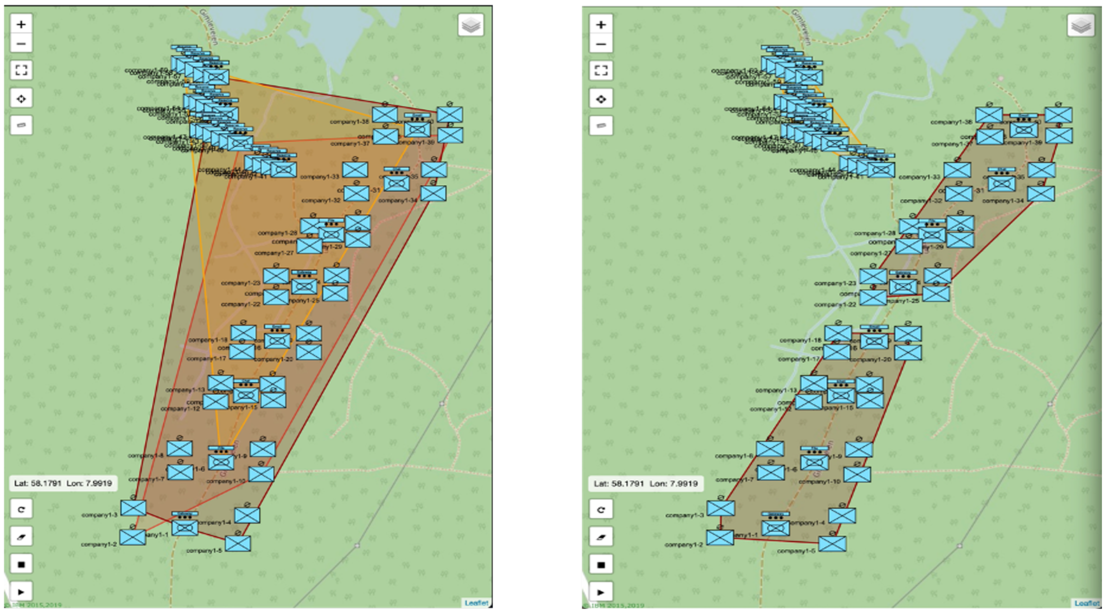


Figure 9.10: Initial and Final Configurations

performed in the rest of the scenario. The initial steps are as follows:

1. The workflow vector is injected by Company1-1, which is on Channel 1. In the demonstration video, this action can be seen in the map display by the multicast of the vector to all assets on Channel 1 and a corresponding transmission of 770bytes. The initial unbound vector is an instruction for Company1-2 to switch to Channel 1. Since we are using dynamic vector truncation, the workflow vector only requires 770 bytes compared to the 1403 bytes that would be required for the corresponding 10Kbit vector message.

2. Company1-2, is initially on Channel 2, so it does not see the initial request and there is no response.

3. Company1-1 does not receive a response on Channel 1 and so switches to Channel 2 and tries again. This time it receives a response from Company1-2 and replies with an acknowledgement vector and switches back to its original channel, which is Channel 1.

4. Company1-2 now has control and switches to the instructed channel, which is Channel 1, and then unbinds the workflow vector and transmits this on Channel 1. The unbound workflow vector in this case is a request for Company1-6 to move to Channel 1.

5. Since Company1-6 is on channel 3, there is no response to the request on Channel 1, and so Company 1-2 switches to Channel 2 and transmits the vector, but again there is no response.

6. Since Company 1-2 has now tried on all of its available channels it needs to discover an asset that can switch to channels that it cannot reach (in this case channel 3).

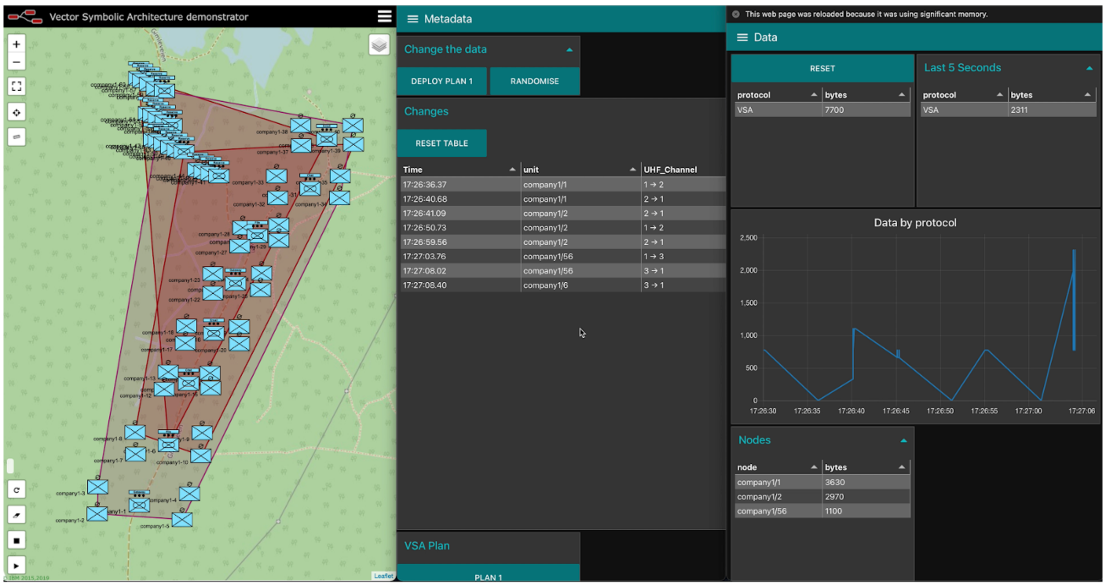
7. Company 1-2 therefore issues a 'Take-Over' request vector on Channel 2 and receives a response from Company1-56 which is a vehicle and can access all three channels.

8. Company 1-2 sends an acknowledgement to Company1-56.

9. Company1-56 now takes over control, switches to Channel 3 and transmits the vector to which Company1-6 responds and Company1-56 acknowledges. This results in Company 1-56 returning to Channel 1 and Company 1-6 switching to Channel 1 as instructed.

The scenario proceeds until all of the assets are on the correct channel. Figure 9.11 shows the state of the demonstration at step 7 in the scenario. The data transmissions and bandwidth consumed were measured both with and without vector truncation respectively. Without vector truncation the total bandwidth required was 252,515 bytes and with truncation was 92,846 bytes, a saving of 2.7x which is significant in the low bandwidth (16.5 Kbit/sec) TacCIS environment in which we are operating.

The total time taken to switch all 20 assets to the correct channel was between 5 min 30 seconds and 6 min, i.e., 16.5 seconds per channel with an average of 4642 bytes per



This web page was reloaded because it was using significant memory.

Time	unit	UHF_Channel
17:26:36.37	company1/1	1 → 2
17:26:40.68	company1/1	2 → 1
17:26:41.09	company1/2	2 → 1
17:26:50.73	company1/2	1 → 2
17:26:59.56	company1/2	2 → 1
17:27:03.76	company1/56	1 → 3
17:27:08.02	company1/56	3 → 1
17:27:08.40	company1/6	3 → 1

protocol	bytes
VSA	7700

protocol	bytes
VSA	2311

node	bytes
company1/1	3630
company1/2	2970
company1/56	1100

Figure 9.11: State of the scenario demonstrator at point 7

asset. This time could potentially be significantly reduced by a more judicious choice of time delays, bringing it closer to approximately 3 seconds per asset on 16.5 kb/sec channels.

9.1.8 Attribute and Proximity Matching for Federated Mission Networks

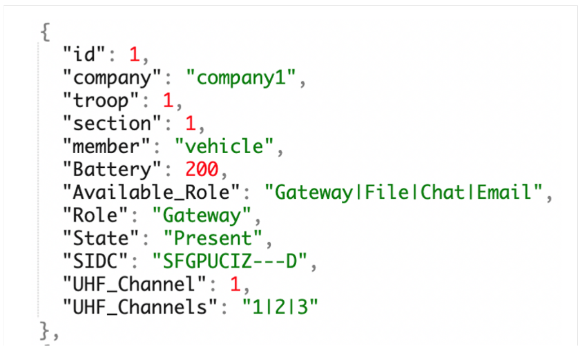
This section describes another separately funded MOD technology transition contract that once again confirms the relevance of the work presented in this thesis.

Federated Mission Networks (FMN) follow the NATO C3 taxonomy of services where each mission network will contain a subset of the possible services and military entities that need to use them in different configurations. There is currently no system that is capable, in an agile way, to automatically discover and connect the services and military entities into new configurations to meet changing mission needs. The objective of this task is to show how shallow semantic attribute matching, using the methods described in Section 5.1 (R1), combined with the proximity based matching described in Section 5.2.1 (R1) can be used to select appropriate services and entities based not only on their functional capabilities but also, on dynamically changing variables such as distance from a specified location.

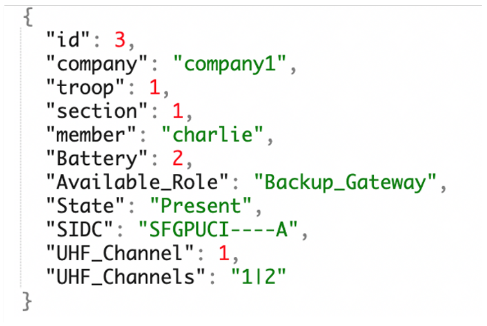
The scenario consists of a number of FMN services (e.g., chat servers, e-mail servers) that are distributed on different assets and which, as in the Radio Reconfiguration scenario, may be operating on different radio channels. The list of assets and their characteristics is given in Table 3.1.

A commander wants to put together a group of assets (vehicles, soldiers) where the assets each have some specified characteristics stored on their associated radio device. For example, the commander wants to issue a VSA workflow that can discover the best-placed chat server and then instruct a collection of team members having the required asset types to connect to this.

To illustrate how the semantic '*number-line*' vectors are used, the commander's request must be constructed such that team members who match the required characteristics (i.e., static node attributes) are discovered but those that are actually chosen are based on their proximity to the chat server. It is important to note that the commander does not need to know the current location and/or radio channel of the required group members in order for the task to be undertaken. The initial location of the assets is shown in Figure 9.14a, and a snapshot of the communications occurring after the workflow has

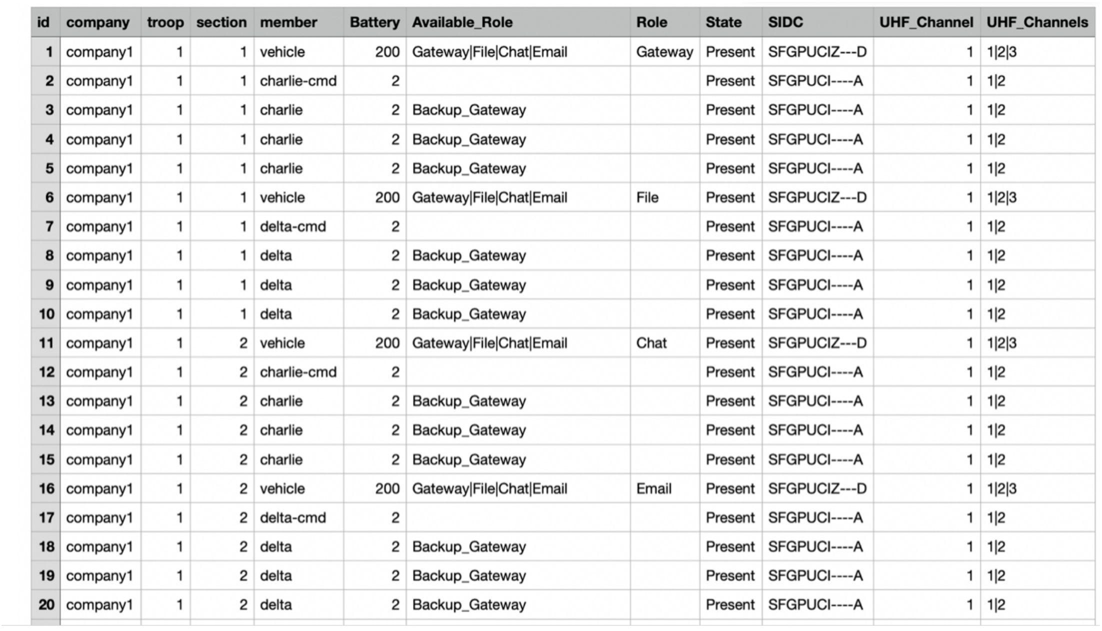


```
{
  "id": 1,
  "company": "company1",
  "troop": 1,
  "section": 1,
  "member": "vehicle",
  "Battery": 200,
  "Available_Role": "Gateway|File|Chat|Email",
  "Role": "Gateway",
  "State": "Present",
  "SIDC": "SFGPUCIZ---D",
  "UHF_Channel": 1,
  "UHF_Channels": "1|2|3"
},
```



```
{
  "id": 3,
  "company": "company1",
  "troop": 1,
  "section": 1,
  "member": "charlie",
  "Battery": 2,
  "Available_Role": "Backup_Gateway",
  "State": "Present",
  "SIDC": "SFGPUCI----A",
  "UHF_Channel": 1,
  "UHF_Channels": "1|2"
}
```

Figure 9.12: JSON representation of two radios each with different configurations.



id	company	troop	section	member	Battery	Available_Role	Role	State	SIDC	UHF_Channel	UHF_Channels
1	company1	1	1	vehicle	200	Gateway|File|Chat|Email	Gateway	Present	SFGPUCIZ---D	1	1|2|3
2	company1	1	1	charlie-cmd	2			Present	SFGPUCI----A	1	1|2
3	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
4	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
5	company1	1	1	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
6	company1	1	1	vehicle	200	Gateway|File|Chat|Email	File	Present	SFGPUCIZ---D	1	1|2|3
7	company1	1	1	delta-cmd	2			Present	SFGPUCI----A	1	1|2
8	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
9	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
10	company1	1	1	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
11	company1	1	2	vehicle	200	Gateway|File|Chat|Email	Chat	Present	SFGPUCIZ---D	1	1|2|3
12	company1	1	2	charlie-cmd	2			Present	SFGPUCI----A	1	1|2
13	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
14	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
15	company1	1	2	charlie	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
16	company1	1	2	vehicle	200	Gateway|File|Chat|Email	Email	Present	SFGPUCIZ---D	1	1|2|3
17	company1	1	2	delta-cmd	2			Present	SFGPUCI----A	1	1|2
18	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
19	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2
20	company1	1	2	delta	2	Backup_Gateway		Present	SFGPUCI----A	1	1|2

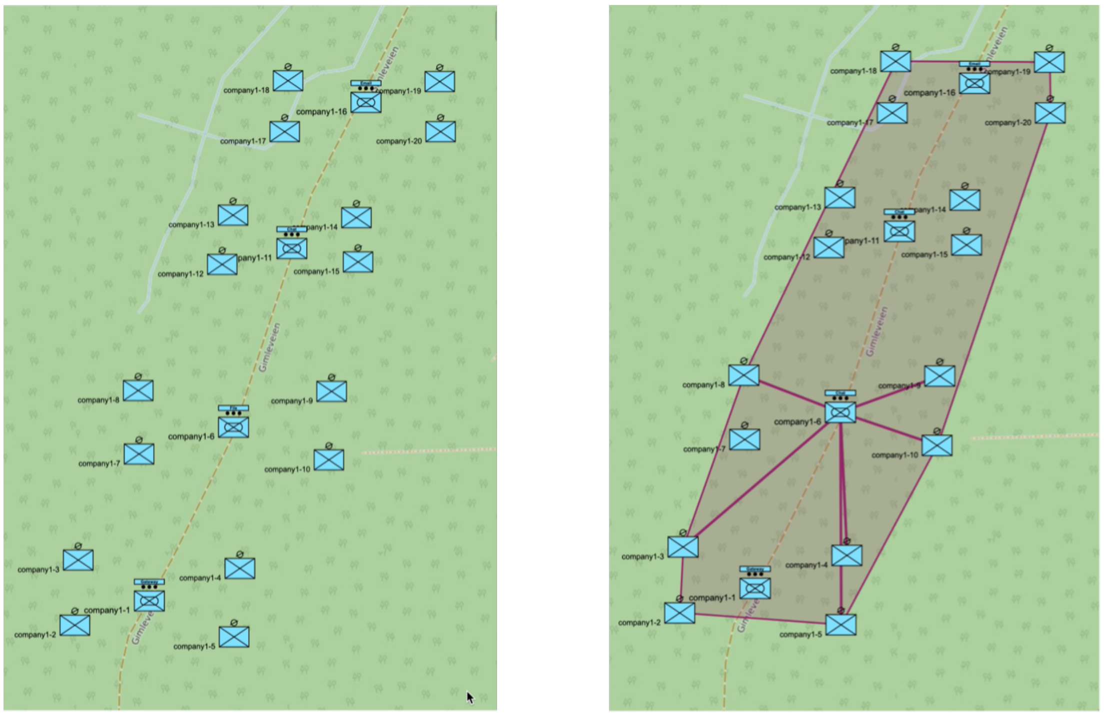
Figure 9.13: List of assets and their characteristics

been executed is shown in Figure 9.14b, where it can be seen that a chat server has been discovered and selected and assets of the required type are now communicating via the discovered chat server.

Experimental Results

In the demonstration scenario, twenty assets are distributed in unknown geographical positions. Each asset creates its SV description from its JSON node description file (R1), see Figure 9.12, and can determine its Military Grid Reference System (MGRS) geospatial reference location from the experimental environment.

The workflow vector is constructed using Eq. 6.1 as a sequence of the nodes description SVs, $\boldsymbol{Z_i}$, where each $\boldsymbol{Z_i}$ is built from a the required node attributes, (i.e., a subset of the attributes shown in Figure 9.12). The required position vector is added to the workflow vector $\boldsymbol{Z_x}$ as metadata before the workflow vector is multicast injected into the network. The VSA enabled nodes match against their static SV descriptions and, if a match is found, weight their HDS score for proximity using the method described



(a) before (b) after

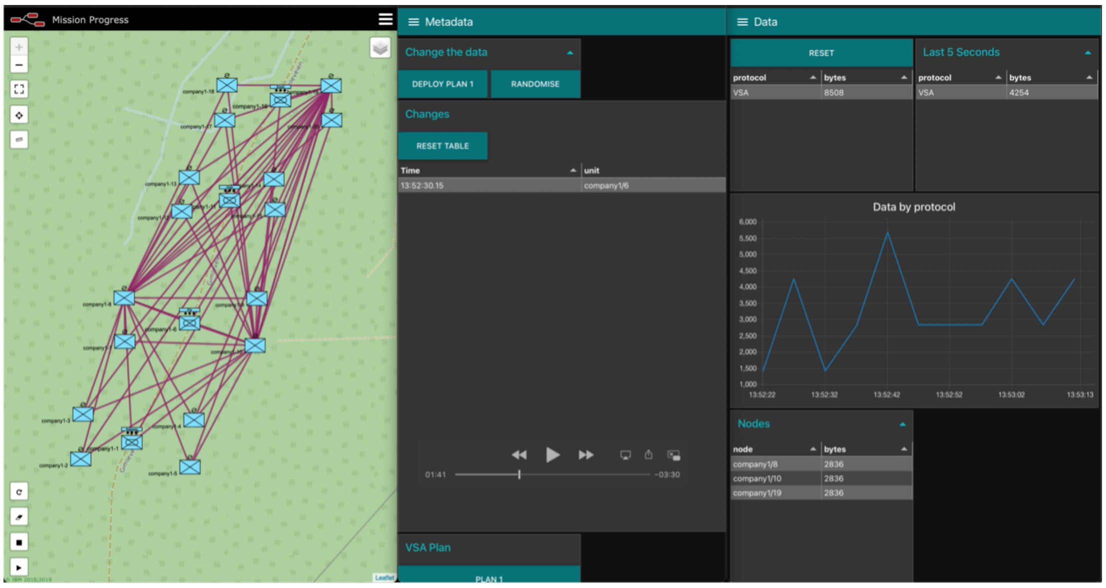
Figure 9.14: Scenario laydown of assets before and after the workflow discovery and connection has been executed.

in Section 5.2.1 (R1).

The demonstrator interface is identical to that used for the Radio Reconfiguration task and is shown in Figure 9.15. The interface shows the location of the assets on a map and the current patterns of communication between the nodes. A series of additional windows together indicate node activity and the volume of communications that are occurring between the different nodes as the scenario unfolds. A video of the activities is available at:

`https://drive.google.com/file/d/1oe7LmVA48c5q6LY0KMSvQJbytpy7u8e6/view?usp=sharing`

This video should be downloaded and viewed at full resolution using slider bar to inspect the state of the scenario at different times during the execution to understand the details of what is occurring.



Time	unit
13:52:30.15	company1/6

protocol	bytes
VSA	8508

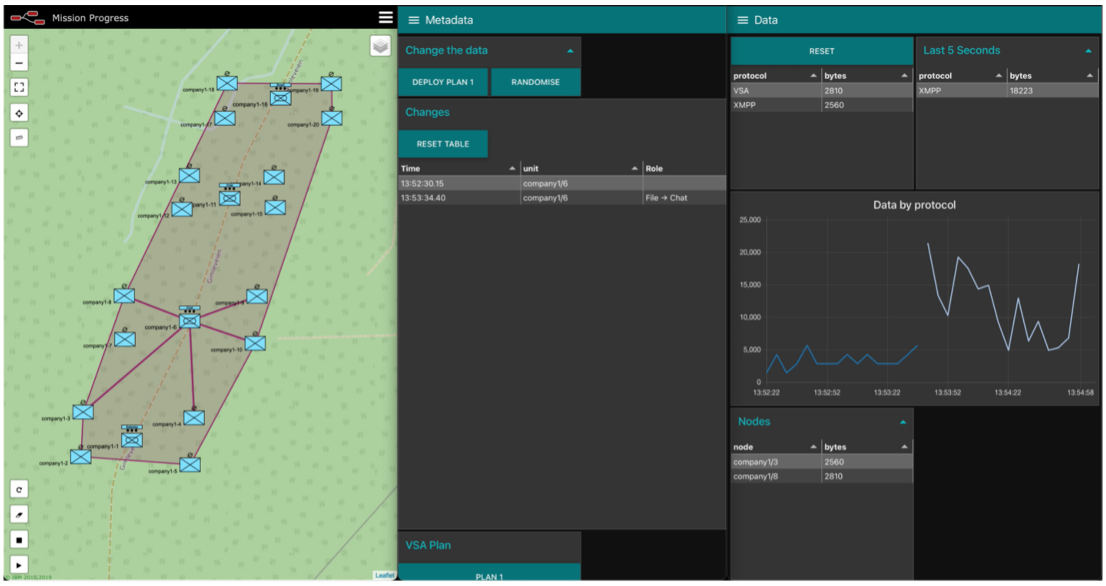
protocol	bytes
VSA	4254

node	bytes
company1/8	2836
company1/10	2836
company1/19	2836

Figure 9.15: The demonstrator interface showing the location of the assets and the communications occurring as the demonstration unfolds.

In Figure 9.16, the demonstrator Interface shows the location of the assets and the communications occurring as the nodes are recruited (blue line in Data Protocol window) and following the completion of the workflow steps when the nodes are communicating via the chat service (white line in the Data Protocol Window). The node recruitment takes approximately 60 seconds with an average network bandwidth utilization 10Kbits/sec in a 12.5 Kbit/sec network.

It should be noted that in this scenario, when selecting the chat server to use, the workflow vector was constructed to prefer a node near to the node initiating the query that had the capability to become a chat server rather than a node having an active chat server if such a node was further away. If the chat server workflow recruitment step omitted the *Available_Role* key-value and instead specified the current *Role* key-value, then the workflow vector would select the closest active server.



Time	unit	Role
13:52:30.15	company1/6	
13:53:34.40	company1/6	File → Chat

protocol	bytes
VSA	2810
XMPP	2560

protocol	bytes
XMPP	18223

node	bytes
company1/3	2560
company1/8	2810

Figure 9.16: The demonstrator Interface showing the location of the assets and the communications occurring as the nodes are recruited (blue line in Data Protocol window) and following the completion of the workflow steps when the nodes are communicating via the chat service (white line in the Data Protocol Window).

9.2 Evaluation of Use Cases

9.2.1 Test Data

I found the use of Shakespeare's Hamlet as a model for constructing scalable hierarchical workflows to be a very useful approach that was easily verifiable. However, during presentations of my work, and despite repeated and careful attempts at clarifying the analogy, a number of the military advisors representing my sponsors did not find the Hamlet workflow analogy to be a convincing representation of real world military scenarios. With this in mind, I conclude that careful selection of test data to reflect believable scenarios for the target audience is an important consideration to be carried forward into the future.

The diversity of DAG workflows available using the Pegasus workflow generator allowed the VSA representation of DAGs to be easily tested and verified. However, the use of stub processes in place of real Pegasus modules should be improved upon in order to test VSA workflows in the most realistic scenarios possible.

9.2.2 Data Collection and Verification

Using a HTTP logger to collect output from the distributed services into a single log file was essential for the empirical analysis, verification and debugging of decentralized P2P workflows.

9.2.3 Dynamic VSA Workflows and QoS - Hamlet workflow

The evaluation was performed using the CORE/EMANE network emulator to simulate a MANET network and used a MANET multicast routing protocol to communicate vectors between the nodes containing the services. Multicasting the top level Hamlet vector results in the whole play being enacted by worker services that generate each word in the play. The VSA workflow implementation of Hamlet has a number of advantages over the Newt[116] implementation. Specifically, the Newt implementation requires that the IP address of participating services be known and encoded into the workflow, whereas our VSA approach can discover the service(word/sentence) needed on the fly using semantic matching. In Newt, if the service specified by IP address becomes unavailable, i.e., we intentionally move it out of wireless range in CORE, then the workflow halts and is broken. In VSA Hamlet, the same action results in the automatic discovery of multiple exact and near-match candidate word/sentence/services, and the best match is then chosen. When multiple, functionally equal matches were discovered, the *local arbitration* function ensured that the service having best simulated utility was chosen and logged as such. The best *'near'* match was chosen when we contrived to make exact matches unavailable in CORE. Additionally, the advantage of passing around the workflow as a vector superposition was highlighted because the stand-in service automatically resynchronized the workflow after *'speaking'* its substitute word by simply performing an unbind and transmit of the workflow vector it received. Newt has none of these capabilities.

9.2.4 Static VSA Workflows - Pegasus DAG workflows

This experiment shows that DAG workflows can be composed, discovered and connected in a completely decentralized way using the VSA hierarchical binding scheme described in section 6.2.4 (R5). The encoding of DAG node descriptions into BSC vector descriptions enables services to be discovered and recruited into the DAG by matching their functional and QoS characteristics to the workflow's vector node descriptions instead of the need to specify the IP-location of each node. By use of semantic vector descriptions for node discovery, this experiment also demonstrated that it is possible to locate alternate similar services which is a significant advantage for environments where network fragmentation is common-place. However, recovery from node disconnects and failures after DAG workflow execution has been started was not addressed. This is significantly more problematic for DAG workflow operation than it is for sequential workflows (where the failing step can be simply restarted) because of the problem associated with how to synchronise the overall workflow state of the DAG. Therefore, recovery from node failures during decentralized execution of DAG workflows should be considered an important and challenging future work area.

While NEWT [62] has demonstrated workflow orchestration of DAG workflows in a decentralized environment, it requires the specification of each node's IP-address location and so would be inflexible to network fragmentation. Therefore, this approach extends the state-of-the-art by being the first to demonstrate flexible DAG discovery and connection in such distributed decentralized environments without the need to specify node address locations.

9.2.5 Node-RED Integration - Traffic Congestion workflow

This experiment successfully demonstrated that Node-RED workflows can be migrated to operate in a decentralized execution environment by converting Node-RED service objects into semantically comparable BSC vectors (R1) and converting Node-RED JSON workflow descriptions (Node-RED *flows*) into my hierarchical VSA workflow representation (R2). This was tested using an existing Node-RED traffic counting workflow and a simulation of a Pegasus DAG workflow. The log output and graphic provided in the video results clearly demonstrate research contributions (R1), (R2), (R4) and (R5) in action, for example:

- **PMATCH: 0.6427 | node_name:** and **PWORK: 0.6427 | node_name:** entries demonstrate (R1). In many sections in the video, for example ($\mathbf{03:04}$) (see also, Fig 9.4), multiple nodes report their HDS match values. This shows that similar services can be semantically compared and differentiated using their BSC SV representations built the $\boldsymbol{json\_to\_vecs}$ method.
- **MATCH: 0.6302 | node_name:** and **WORK: 0.6136 | node_name:** entries demonstrate (R1). The service object chosen will have the the highest HDS score, which can be checked by comparing to preceding PMATCH and PWORK log entries.
- **%TAKEID%** entries demonstrate the recruit phase of the VSA DAG encoding Section 6.2.4 (R2). The winning node can be seen to correctly calculate its activation position within the workflow.
- **%REQCON%** and **%TAKCON%** demonstrate the connection phase of the VSA DAG encoding, Section 6.2.4 (R2).
- Figure 9.4 and ($\mathbf{03:04}$) clearly demonstrate (R4). Multiple nodes actively cancel their planned transmission on seeing a better match.

The Node-RED graphical user interface is very intuitive and easy to use. By integrating my VSA framework with Node-RED, this use case has demonstrated that Node-RED could find new employment for rapid workflow creation in military field operations where the network address of assets is difficult to keep track of due to frequent fragmentation.

9.2.6 Binary Spatter Code Message Truncation (R5)

This experiment was an empirical evaluation of the mathematical model presented in Chapter 7. The results confirm that "Holographic dynamic message sizing (R4)" is an effective method to reduce the size of BSC message packets for on-the-wire communication, thereby reducing communication bandwidth while maintaining the semantic information content. From the test-case results, it is noted that while the resulting bandwidth savings may appear low in terms of MB saved and would not be important in a fixed network infrastructure, savings of 45% in our target environment will prove to be extremely important. This is because, for tactical edge military networks, bandwidth can become a critical resource when devices are mobile. At greater distances from their nearest neighbours, bandwidth is increasingly reduced due to *path loss* and networks become fragmented. This thesis is the first to suggest using the holographic properties of the symbolic vectors to perform compression, taking into consideration the number of combined sub-vectors along with similarity bounds that determine conflict with other encoded vectors used in the same context.

9.2.7 Radio Reconfiguration with Neuromorphic Phase-change Memory Integration

This technology transition task demonstrated how my VSA workflow framework could be used to solve a real-world military problem that currently does not have a satisfactory solution using other means. My VSA framework was used to successfully perform a communications replanning task in a Tactical Communications and Information Systems (TacCIS) environment. The communications plan was represented as a BSC workflow vector (R2) using eq. (6.1) which, when multicast injected into the radio network, was exchanged between assets to discover and instruct assets on different radio channels to reconfigure into a desired configuration (communications plan). The associated demonstration environment shows that the reconfiguration of twenty assets, initially assigned to random communication channels, can be discovered and instructed to achieve this goal in a completely decentralized manner (R5). Whilst there is still scope for improvements in terms of performance, the feasibility of such an approach has been demonstrated.

This use case clearly demonstrates the effectiveness of my research contributions, (R2), (R3), and (R5) in a number of ways:

- (R2) is demonstrated because it was easy to construct a single BSC workflow vector that was passed around from peer to peer in away that caused each node to reprogram its radio communication channel onto a new designation.
- The successful execution of *take-over* requests also demonstrated the effectiveness of representing workflows as BSC vectors (R2) because it allowed for any node executing the *take-over* to re-synchronise its cached copy of the workflow vector in question into the correct/active workflow step permutation in a very simple manner.
- The effectiveness of a one-to-many reactive communications paradigm, "cognitive workflow (R5)", is demonstrated by successful execution of *take-over* requests because it enables any node to cache any workflow vector it '*hears*' on the multicast channel, regardless of whether or not the node is participating in the workflow. This enables nodes to perform *take-over* requests in a very simple manner and without the need to understand the information content of the workflow for which it is proxying.

- The effectiveness of "Holographic dynamic message sizing (R3)" was confirmed because the experiment reported a saving of 2.7x reduction in bandwidth consumption (252,515 bytes without truncation, compared to 92,846 bytes with truncation). This is a significant saving for the low bandwidth (16.5 Kbit/sec) TacCIS environment.

Phase Change Memory Results

Operation of the PCM device was benchmarked against a conventional 65 nm CMOS device, see [6, Section 4.2]. The results found that the PCM device had a total energy efficiency of 117.5:1 compared to the all CMOS configurations. Similarly there was a 31.9 x reduction in required chip area. This suggests that these types of devices would clearly have a significant impact in energy constrained environments such as IoBT and IoT applications. From the perspective of this thesis, this is a strong argument for the use of BSC for the representation, discovery and orchestration of services and workflows because PCM devices are inherently parallel processing architectures that are designed to operate on high-dimensional vectors.

9.2.8 Attribute and Proximity Matching for Federated Mission Networks

This transition task is a compelling demonstration of (R1), (R2), and (R5) because it successfully demonstrated how my VSA framework could be used to perform agile command-and-control tasks in a TacCIS type environment fulfilling a so far unsolved MOD requirement. The task successfully demonstrated resource discovery using vector encoded attribute collections, Section 5.4 (R1), weighted by the resource's position relative to a request position, both encoded as 2D *number-line* vectors as described in Section 5.2.1 (R1). A sequence of the required vector asset descriptions is encoded into a BSC workflow request (asset recruitment) vector (R2), along with the target position vector (added as metadata). When multicast injected into the network, the workflow request is executed as described in, sections 6.2.2 (R2) and 8.1 (R5), so that the resources are discovered and connected in a completely decentralized manner.

My VSA workflow framework could therefore provide an entirely new approach to agile command and control. Mission goals can be achieved by discovering resources based on their functional capabilities (semantic attribute matching) and dynamically changing operational characteristics such as vehicle fuel status, radio battery life, etc. Such an approach would optimise the use of available assets while minimising the need for detailed pre-mission planning.

Chapter 10

Conclusion

For wired networks, there are many successful WFMS available [18–27]. However, in the current state-of-the-art, workflows and workflow management systems are heavily reliant on centralized architectures using stable high bandwidth networking and must be specified imperatively, as described in Sections 1.3 and 3.1. More recently, research into Edge computing [65–67] is focused on exploiting the processing power available in edge devices to reduce loads on centralized controllers, improve latency and reduce bandwidth consumption. In both of these approaches, stable connectivity among edge devices and to cloud servers is assumed, as is a known network address location for each device.

When workflows must operate in more dynamic environments, such as MANETs (as described in section 3.2), service discovery becomes an essential component of the WFMS. Current approaches for service discovery in MANETs are classified as pro-active directory based or reactive directory-less. Directory based architectures, which must be distributed for useful operation in MANETs, carry more bandwidth overhead due to the necessity of server announcements and requirement to keep directories up to date. This is an issue in low-bandwidth scenarios. In addition, because directories can easily go out of service during network fragmentation, directory based approaches are not appropriate when networks are likely to suffer frequent fragmentation, as is the case for military field operations. Directory-less architectures, which typically use broadcast or multicast, are more flexible in the face of network fragmentation. However, this approach can cause significant network congestion [74] and therefore present difficulties for operation in low-bandwidth networks.

In the current state-of-the-art, the important issue of matching requests to services is implemented in a number of ways: 1) mapping known service identifies to URLs, 2) attribute matching and, in a few cases, 3) ontology matching techniques [75–77]. Each approach has some benefits and weaknesses. Simple identifier approaches keep workflow discovery message overhead low and can often be embedded in the underlying transport protocol such as in DNS-SD architectures, however, fuzzy semantic matching is not possible. Attribute matching allows for richer queries, but attribute names must be agreed and assessing match quality can be computationally difficult and brittle [93–95] as discussed in section 3.3.1. Ontologies such as OWL-S and W3C can be used for semantic matching but, as described in section 3.3.1, they are difficult to develop and maintain, especially when targeted for use among loosely cooperating coalition partners. They are not easily combined or extended and can become large and unwieldy (which is not ideal for low power mobile devices).

Therefore, a need is identified for a compact representation that can facilitate complex semantic matching on workflow requests with little overhead and is capable of supporting the execution of workflows in a decentralized, low-bandwidth, transient environment without reliance on central registries and centralized controllers.

This thesis has shown that the basis of such a workflow architecture can be formulated using a Binary Spatter Code (BSC) Vector Symbolic Architecture (VSA). Resource and service objects build and maintain BSC vector representations of themselves, enabling fuzzy matching to workflow requests via a simple Hamming similarity test. Complex workflows are also represented as BSC vectors that contain the details needed for composition, discovery and orchestration of the services and resources needed to complete a task. The one-to-many reactive workflow paradigm facilitates cooperation between autonomous decentralized agents (service objects, resources and sensors) while the decentralized local arbitration mechanism minimizes the number of workflow orchestration messages that must be exchanged to control workflow execution. In addition, because BSC vectors are distributed representations, further bandwidth optimization is obtained through vector truncation while maintaining full matching and operational capability.

The research contributions are now discussed commenting on their impact and potential to fulfil existing issues with respect to the current state-of-the art:

Research contribution R1 - Multi-modal semantic service object descriptions:

Alternate approaches for the description and semantic matching of service objects were considered in Section 3.3. Complex semantic matching for service discovery is typically performed using a service ontology such as OWL-S and W3C. However, ontologies are difficult to develop and maintain, especially when targeted for use among loosely cooperating coalition partners and require complex processing to perform semantic matching.

Representing services and resources as vectorized attribute collections, Chapter 5, represents an improvement in the state-of-the-art for service discovery, particularly when considering low power edge devices, because semantic comparisons can be made using a computationally efficient Normalised Hamming Similarity (HDS) calculation, especially if the calculation is implemented in hardware. In addition, by using BSC to represent services, rich multi-model descriptions of service objects and sensors can be created by encoding the object's sub-feature vectors from a combination of simple VSA *binding* and *bundling* operations as well as from existing semantic vector word databases (Section 5.3), and non-text based vector descriptions using the method of randomized binary projection [108]. Matching can also be performed on approximate values using the range and 'number-line' methods, Section 5.2.1. Binding and bundling operations are mathematically well defined, and therefore, similarity comparisons are deterministic and explainable.

Research contribution R2 - Hierarchical VSA bundling:

I extend VSAs using a novel hierarchical vector binding and bundling scheme that maintains the ability to perform semantic matches and avoids the false activation and nullification of embedded sub-vectors that plague previous schemes. The scheme is capable of recursively bundling multiple levels of abstraction (workflow and sub-workflows/branches) to a practically unlimited depth as demonstrated by the Hamlet use case 9.2.3. The ability to scale is a major advantage since it is not unusual for workflows to require many hundreds of individual steps in today's microservice workflow architectures. Describing workflows using a self-contained BSC vector provides a very efficient way of controlling decentralized workflow execution because the vector can be '*wound*' forwards and backwards to any position in the workflow without needing knowledge of the workflow vector's operational content as evidenced in the Radio Reconfiguration use case when any node was able to act as a proxy for an unrelated node during the processing of *takeover-request*, see Sections 9.1.7 and 9.2.7, as well as in the Hamlet use case, Section 9.2.3, which showed that the dynamic workflow automatically re-synchronised onto its original path after an alternate service was selected. The new encoding encapsulates all the information needed to control workflow discovery and execution without a central controller. It also provides a unique context for every workflow execution step, which has the potential to enable service agents to learn the meaning of other agents (and themselves) in a similar way to how word meanings are learnt in NLP, see Section 6.2.6.

Research contribution R3 - Holographic dynamic message sizing:

Chapter 7 extends the state-of-the-art for BSC vector symbolic research in general by being the first to provide a definitive mathematical model for calculating the minimum vector dimension needed to avoid confusion when trying to differentiate between BSC vectors that are similar. Message size optimization is particularly important for the edge computing in low-bandwidth transient MANET environments, and its effectiveness was demonstrated in the Radio Reconfiguration task, 9.1.7 and 9.2.7 when it achieved a 2.7x reduction in consumed bandwidth compared to workflow execution without truncation.

Research contribution R4 - Bandwidth efficient distributed arbitration:

The multicast reactive model for workflow discovery is the simplest to manage, but reactive multicast service discovery protocols are prone to network congestion when many servers and requesters are issuing announcements and requests [74]. The delayed response mechanism, Section 8.2.5, extends the-state-of-the-art in service discovery because it will enable the multicast reactive model to be used without causing network congestion. When combined with the representation of services as BSC vectors (R1), it is a very efficient mechanism because all listeners have a very simple method to test for semantic match quality, using hamming similarity, and therefore, they are not overloaded processing match calculations in order to make decisions on their delayed response.

Research contribution R5 - Cognitive workflow model:

Section 3.1 identifies that WFMS are almost exclusively controlled by a centralized task coordinator/manager. This means that they are not suitable for operation in MANET environments where fragmentation is frequent and stable end points cannot be guaranteed. This thesis extends the workflow state-of-the-art by using a reactive one-to-many communication model for all workflow orchestration messages, represented as BSC vectors. This means that all workflow orchestration messages can be received and acted upon by any listening service, which enables services to gain awareness of workflow activity in the network and facilitates cooperation between autonomous services as demonstrated in the Radio Reconfiguration use case when any node was able to act as a proxy for an unrelated node during the processing of *takeover-request*, see sections 9.1.7 and 9.2.7. Combined with (R4), which minimizes workflow orchestration message overhead, this creates a truly peer-to-peer architecture capable of operating in transient MANET environments without a central point of control.

This thesis is the first work to describe workflows and their component services as distributed vector representations. The outcomes demonstrate a new cognitive workflow model that uses one-to-many communications to enable intelligent cooperation between self-describing service entities that can self-organise to complete a workflow task. Workflow orchestration overhead was minimized using two innovations, a local arbitration mechanism that uses a delayed response mechanism to suppress responses that are not an ideal match and the holographic nature of VSA descriptions enables messages to be truncated without loss of meaning. A new hierarchical VSA encoding scheme was created that is scaleable to any number of vector embeddings including workflow steps. The encoding can also facilitate learning since it provides unique contexts for each step in a workflow. The encoding also enables service pre-provisioning because individual workflow steps can be decoded easily by any service receiving a multicast workflow vector.

10.1 Future Work

10.1.1 How can semantic representation of service objects be improved?

As described in Section 5.5, the encoding methods described in Sections 5.1 and 5.4 for the creation of Service Vector (SV) descriptions are effectively vector representations of unordered collections of key-value attribute pairs. As such, these type of SV encodings can only offer shallow semantic, i.e., syntactic matching, capability. A further work objective is to consider how deeper semantic representations can be captured. This is a particularly challenging requirement because very often true semantic similarity depends on context. For example, consider the challenge of locating a camera sensor near a particular location, say the north end of Oxford Street. We can specify a lat-long as a key-value entry in a JSON camera resource description file and use the proximity matching technique described in Section 5.2.1 to encode the location. If the objective of the workflow is a search for some vehicle near the location specified, then any camera sensors returned near the lat-long specified, including cameras in adjacent streets, might be considered a good match. However, if the workflow objective is to obtain video analysis of an incident specifically happening at the north end of Oxford street, then only cameras with a view of Oxford Street would be acceptable. Thus, in the second scenario a camera located half a mile further down Oxford street might be an acceptable match but very close cameras in adjacent streets would not.

Another might be to explore how query by example could be encoded in SV descriptions. That is, is it possible in some way to capture the transfer function of a microservice, considered as a black box, by encoding examples of the typical input it accepts and corresponding output it produces?

The fundamental research question might be how to create deep semantic vector representations of services and resources?

10.1.2 Learn a semantic vector space of workflows

Word2Vec [106, 107] creates semantic vector space for words by processing a large corpus of text. The resulting vectors can be combined and operated on in ways that make sense to the human brain, for example $\boldsymbol{KING} - \boldsymbol{MALE} = \boldsymbol{QUEEN}$. As described in Section 6.2.6, the hierarchical binding scheme described in eq. (6.1) creates unique vector contexts in the workflow vector, eq. (6.4) and eq. (6.6), as workflow execution proceeds. A fundamental research question might be:

Can these unique contexts be used to learn a semantic vector space of workflows that would allow such semantic workflow vectors to be combined in order to create new and novel workflows that have not been executed before?

Bibliography

[1] Chris Simpkin, Ian Taylor, Graham A Bent, Geeth De Mel, and Swati Rallapalli. Decentralized microservice workflows for coalition environments. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 2017.

[2] Christopher Simpkin, Ian Taylor, Graham A Bent, Geeth de Mel, and Raghu K Ganti. A scalable vector symbolic architecture approach for decentralized workflows. In *COLLA 2018 The Eighth International Conference on Advanced Collaborative Networks, Systems and Applications*, pages 21–27. IARIA, 2018.

[3] Christopher Simpkin, Ian Taylor, Daniel Harborne, Graham Bent, Alun Preece, and Ragu K Ganti. Dynamic distributed orchestration of node-red iot workflows using a vector symbolic architecture. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 52–63. IEEE, 2018.

[4] Chris Simpkin, Ian Taylor, Graham A Bent, Geeth de Mel, Swati Rallapalli, Liang Ma, and Mudhakar Srivatsa. Constructing distributed time-critical applications using cognitive enabled services. *Future Generation Computer Systems*, 100:70–85, 2019.

[5] Chris Simpkin, Ian Taylor, Daniel Harborne, Graham Bent, Alun Preece, and Raghu K Ganti. Efficient orchestration of node-red iot workflows using a vector symbolic architecture. *Future Generation Computer Systems*, 111:117–131, 2020.

[6] Graham Bent, Christopher Simpkin, Ian Taylor, Abbas Rahimi, Geethan Karunaratne, Abu Sebastian, Declan Millar, Andreas Martens, and Kaushik Roy.

Energy efficient 'in memory' computing to enable decentralised service workflow composition in support of multi-domain operations. In Tien Pham and Latasha Solomon, editors, *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III*, volume 11746, pages 414 – 435. International Society for Optics and Photonics, SPIE, 2021. doi: 10.1117/12.2586988. URL `https://doi.org/10.1117/12.2586988`.

[7] CERN: Facts and figures about the LHC. `https://home.cern/resources/faqs/facts-and-figures-about-lhc`.

[8] Twitter usage statistics. `https://www.internetlivestats.com/twitter-statistics/`.

[9] Google usage statistics. `https://www.internetlivestats.com/google-search-statistics/#trend/`.

[10] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *science*, 332(6025):60–65, 2011.

[11] Martin Hilbert. Quantifying the data deluge and the data drought. *Available at SSRN 2984851*, 2015.

[12] BBVA OpenMind. The top 10 supercomputers, the new scientific giants. `https://www.bbvaopenmind.com/en/technology/innovation/the-top-10-supercomputers-the-new-scientific-giants/`.

[13] Stefan M Larson, Christopher D Snow, Michael Shirts, and Vijay S Pande. Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology. *arXiv preprint arXiv:0901.0866*, 2009.

[14] Wikipedia contributors. Folding@home — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Folding@home&oldid=952201358`, 2020. [Online; accessed 2020-04-21].

[15] Frank J Seinstra, Jason Maassen, Rob V Van Nieuwpoort, Niels Drost, Timo Van Kessel, Ben Van Werkhoven, Jacopo Urbani, Ceriel Jacobs, Thilo Kielmann, and Henri E Bal. Jungle computing: Distributed supercomputing beyond clusters, grids, and clouds. In *Grids, Clouds and Virtualization*, pages 167–197. Springer, 2011.

[16] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[17] Majid Hajibaba and Saeid Gorgin. A review on modern distributed computing paradigms: Cloud computing, jungle computing and fog computing. *Journal of computing and information technology*, 22(2):69–84, 2014.

[18] Marek Wieczorek, Radu Prodan, and Thomas Fahringer. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Record*, 34(3): 56–62, 2005.

[19] T. Fahringer, R. Prodan, R.Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. *Workflows for e-Science*, chapter ASKALON: A Development and Grid Computing Environment for Scientific Workflows, pages 143–166. Springer, New York, 2007.

[20] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 423–424. IEEE Computer Society, New York, 2004.

[21] Peter Kacsuk. P-grade portal family for grid infrastructures. *Concurr. Comput. : Pract. Exper.*, 23:235–245, March 2011. ISSN 1532-0626. doi: http://dx.doi.org/10.1002/cpe.1654. URL `http://dx.doi.org/10.1002/cpe.1654`.

[22] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D.S. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[23] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[24] Andrew Harrison, Ian Taylor, Ian Wang, and Matthew Shields. WS-RF Workflow in Triana. *International Journal of High Performance Computing Applications*, 22(3):268–283, August 2008. ISSN 1094-3420. doi:

10.1177/1094342007086226. URL `http://hpc.sagepub.com/cgi/doi/10.1177/1094342007086226`.

[25] Roger Barga, Jared Jackson, Nelson Araujo, Dean Guo, Nitin Gautam, and Yogesh Simmhan. The trident scientific workflow workbench. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 317–318, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3535-7. doi: 10.1109/eScience.2008.126. URL `http://dl.acm.org/citation.cfm?id=1488725.1488936`.

[26] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR. *Int. J. High Perform. Comput. Appl.*, 22:347–360, August 2008. ISSN 1094-3420. doi: 10.1177/1094342008096067. URL `http://dl.acm.org/citation.cfm?id=1400050.1400057`.

[27] Bartosz Balis. Increasing scientific workflow programming productivity with hyperflow. In *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*, WORKS '14, pages 59–69, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-7067-4. doi: 10.1109/WORKS.2014.10. URL `http://dx.doi.org/10.1109/WORKS.2014.10`.

[28] Workflow Management Coalition. The Workflow Management Coalition. `http://www.wfmc.org/`.

[29] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. In *CLOSER*, pages 221–232, 2018.

[30] Stephen Russell and Tarek Abdelzaher. The internet of battlefield things: the next generation of command, control, communications and intelligence (c3i) decision-making. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 737–742. IEEE, 2018.

[31] S. Basagni, M. Conti, S. Giordano, and I. Stojmenović. *Mobile Ad Hoc Networking: Edited by Stefano Basagni...[et Al.]*. IEEE, 2004.

[32] Georgios Fakas and Bill Karakostas. A peer to peer (p2p) architecture for dynamic workflow management. *Information and Software Technology*, 46:423–431, 05 2004. doi: 10.1016/j.infsof.2003.09.015.

[33] Z Trifa and M Khemakhem. Effects of churn on structured p2p overlay networks. In *Proceedings of International Conference on Automation, Control, Engineering and Computer Science, ACECS*, pages 164–170, 2014.

[34] T. Pham, G. Cirincione, A. Swami, G. Pearson, and C. Williams. Distributed analytics and information science. In *In IEEE International Conference on Information Fusion (Fusion),*, 2015.

[35] Dinesh Verma, Graham Bent, and Ian Taylor. Towards a distributed federated brain architecture using cognitive iot devices. *COGNTIVE 2017*, page 98, 2017.

[36] Tony A Plate. *Distributed representations and nested compositional structure*. University of Toronto, Department of Computer Science, 1994.

[37] Tony A Plate. Holographic reduced representations. *IEEE Transactions on Neural networks*, 6(3):623–641, 1995.

[38] M. N Jones and D. J. K. Mewhort. Representing word meaning and order information in a composite holographic lexicon. *psychological Review*, 114(1): 1–37, 2007.

[39] Gregory E Cox, George Kachergis, Gabriel Recchia, and Michael N Jones. Toward a scalable holographic word-form representation. *Behavior research methods*, 43(3):602–615, 2011.

[40] Gabriel Recchia, Magnus Sahlgren, Pentti Kanerva, and Michael N Jones. Encoding sequential information in semantic space models: comparing holographic reduced representation and random permutation. *Computational intelligence and neuroscience*, 2015:58, 2015.

[41] Tony A. Plate. *Holographic Reduced Representation: Distributed Representation for Cognitive Structures*. CSLI Publications, Stanford, CA, USA, 2003. ISBN 1575864290.

[42] Chris Eliasmith, Terrence C. Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *Science*, 338(6111):1202–1205, November 2012. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.1225266. URL `http://www.sciencemag.org/content/338/6111/1202`.

[43] Chris Eliasmith and Paul Thagard. Integrating structure and meaning: A distributed model of analogical mapping. *Cognitive Science*, 25(2):245–286, 2001.

[44] R.W. Gayler and S.D Levey. A distributed basis for analogical mapping. In *New Frontiers in Analogy Research, Proceedings of the Second International Conference on Analogy, ANALOGY-2009*, 2009.

[45] Chris Eliasmith. *How to build a brain: A neural architecture for biological cognition*. Oxford University Press, 2013.

[46] Denis Kleyko. *Pattern Recognition with Vector Symbolic Architectures*. PhD thesis, Luleå tekniska universitet, 2016.

[47] Benny B Nasution and Asad I Khan. A hierarchical graph neuron scheme for real-time pattern recognition. *IEEE Transactions on Neural Networks*, 19(2): 212–229, 2008.

[48] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2):139–159, 2009. URL `http://dblp.uni-trier.de/db/journals/cogcom/cogcom1.html#Kanerva09`.

[49] Distributed analytics and information science international technology alliance program announcement:. `https://www.arl.army.mil/business/collaborative-alliances/dais-ita/`, 2015. [Online; accessed 2020-09-29].

[50] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[51] Arif Ahmed and Ejaz Ahmed. A survey on mobile edge computing. 01 2016. doi: 10.1109/ISCO.2016.7727082.

[52] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[53] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Workflow Management in GriPhyN. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid Resource*

Management, volume 64 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers, Dordrecht, 2003.

[54] Ian Taylor, Matthew Shields, Ian Wang, and Omer Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[55] Shalil Majithia, Ian Taylor, Matthew Shields, and Ian Wang. Triana as a Graphical Web Services Composition Toolkit. In Simon J. Cox, editor, *Proceedings of UK e-Science All Hands Meeting*, pages 494–500. EPSRC, CD-Rom only, September 2003. URL `http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/113.pdf`.

[56] Node-RED: Flow-based programming for the Internet of Things. `https://nodered.org/`.

[57] Apache Taverna SCUFL2 scripting language. `https://taverna.incubator.apache.org/documentation/scufl2/language/,`.

[58] Taverna Service Discovery Plugin. `http://dev.mygrid.org.uk/wiki/display/developer/Tutorial+-+Service+discovery+plugin`.

[59] Lambert M. Surhone, Mariam T. Tennoe, and Susan F. Henssonow. *Node.Js*. Betascript Publishing, Mauritius, 2010. ISBN 6133180196, 9786133180192.

[60] RabbitMQ is the most widely deployed open source message broker. `https://www.rabbitmq.com/`.

[61] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[62] Joseph P. Macker and Ian Taylor. Orchestration and analysis of decentralized workflows within heterogeneous networking infrastructures. *Future Generation Computer Systems*, 75:388 – 401, 2017. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2017.01.007. URL `http://www.sciencedirect.com/science/article/pii/S0167739X17300262`.

[63] William Shakespeare. *The Tragedy of Hamlet*. University Press, 1904.

[64] Javier Fabra, Pedro Álvarez, José A Bañares, and Joaquin Ezpeleta. Deneb: a platform for the development and execution of interoperable dynamic web processes. *Concurrency and Computation: Practice and Experience*, 23(18): 2421–2451, 2011.

[65] Jiayue Liang, Fang Liu, Shen Li, and Zhenhua Cai. A comparative research on open source edge computing systems. In *International Conference on Artificial Intelligence and Security*, pages 157–170. Springer, 2019.

[66] Apache Edgent: A Community for Accelerating Analytics at the Edge. `https://edgent.incubator.apache.org/`,.

[67] Linux Foundation: The EdgeX Foundry Project. `https://www.edgexfoundry.org/`.

[68] Rahul Khot. Cse6306 jiniâ¢ architecture.

[69] Multicast DNS. `https://tools.ietf.org/html/rfc6762`.

[70] M. Giordano. DNS-Based discovery system in service oriented programming. *Lecture notes in computer science*, 3470:840, 2005.

[71] Daniel Steinberg and Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005. ISBN 0596101007.

[72] Joseph Macker and Ian Taylor. Indi: Adapting the multicast dns service discovery infrastructure in mobile wireless networks. In *2011-MILCOM 2011 Military Communications Conference*, pages 1616–1621. IEEE, 2011.

[73] Kyriakos Manousakis, Sharanya Eswaran, David Shur, Gaurav Naik, Pavan Kantharaju, William Regli, and Brian Adamson. Torrent-based dissemination in infrastructure-less wireless networks. *Journal of Cyber Security*, 4:1–22.

[74] Christopher N Ververidis and George C Polyzos. Service discovery for mobile ad hoc networks: a survey of issues and techniques. *IEEE Communications Surveys & Tutorials*, 10(3):30–45, 2008.

[75] Michael Klein and Birgitta König-Ries. Multi-layer clusters in ad-hoc networksâan approach to service discovery. In *International Conference on Research in Networking*, pages 187–201. Springer, 2002.

[76] Dipanjan Chakraborty, Anupam Joshi, Yelena Yesha, and Tim Finin. Toward distributed service discovery in pervasive computing environments. *IEEE Transactions on mobile computing*, 5(2):97–112, 2005.

[77] Hessam Moeini, I Yen, Farokh Bastani, et al. Summarization in semantic based service discovery in dynamic iot-edge networks. *arXiv preprint arXiv:2009.02858*, 2020.

[78] Frank Van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of knowledge representation*. Elsevier, 2008.

[79] Kerry Trentelman. Survey of knowledge representation and reasoning systems. 2009.

[80] Roberta Calegari, Giovanni Ciatto, and Andrea Omicini. On the integration of symbolic and sub-symbolic techniques for xai: A survey. *Intelligenza Artificiale*, 14(1):7–32, 2020.

[81] ACT-R Research Group: About. `http://act-r.psy.cmu.edu/about/`.

[82] Owl web ontology language. URL `http://www.w3.org/TR/owl-features/`.

[83] The World Wide WebConsortium (W3C) . `https://www.w3.org/`.

[84] Marta Sabou. Building web service ontologies. *SIKS Dissertation Series*, (2004-4), 2006.

[85] Hai Dong, Farookh Khadeer Hussain, and Elizabeth Chang. Semantic web service matchmakers: state of the art and challenges. *Concurrency and Computation: Practice and Experience*, 25(7):961–988, 2013.

[86] R. McCool. Rethinking the semantic web. part i. *IEEE Internet Computing*, 9 (6):88–87, 2005. doi: 10.1109/MIC.2005.133.

[87] R. McCooI. Rethinking the semantic web. part 2. *IEEE Internet Computing*, 10 (1):93–96, 2006. doi: 10.1109/MIC.2006.18.

[88] Andreas Blass, Yuri Gurevich, Efim Hudis, et al. The tower-of-babel problem, and security assesment sharing. *Bulletin of EATCS*, 2(101), 2013.

[89] Andrew Iliadis. The tower of babel problem: making data make sense with basic formal ontology. *Online Information Review*, 2019.

[90] Emhimed Salem Alatrish. Comparison of ontology editors. *eRAF Journal on Computing*, 4:23–38, 2012.

[91] Jeff Heflin and James Hendler. Dynamic ontologies on the web. In *AAAI/IAAI*, pages 443–449, 2000.

[92] Bach Thanh Le, Rose Dieng-Kuntz, and Fabien Gandon. On ontology matching problems. *ICEIS (4)*, pages 236–243, 2004.

[93] Mark A Musen and Johan Van der Lei. Of brittleness and bottlenecks: Challenges in the creation of pattern-recognition and expert-system models. In *Machine Intelligence and Pattern Recognition*, volume 7, pages 335–352. Elsevier, 1988.

[94] Deniz Yuret. The binding roots of symbolic ai: a brief review of the cyc project. *this article takes a critical look at the Cyc system. The disadvantages of such type of control systems have been listed here*, 1996.

[95] Paul Smolensky. Connectionist ai, symbolic ai, and the brain. *Artificial Intelligence Review*, 1(2):95–109, 1987.

[96] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[97] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, Agnese Pinto, and Eugenio Di Sciascio. Machine learning in the internet of things: A semantic-enhanced approach. *Semantic Web*, 10(1):183–204, 2019.

[98] Jiajun Zhang, Chengqing Zong, et al. Deep neural networks in machine translation: An overview. *IEEE Intell. Syst.*, 30(5):16–25, 2015.

[99] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.

[100] Maad M Mijwel. Artificial neural networks advantages and disadvantages. *Retrieved from LinkedIn: https://www. linkedin. com/pulse/artificial-neuralnet works-advantages-disadvantages-maad-m-mijwel*, 2018.

[101] Matthew Kelly and Robert West. From vectors to symbols to cognition: The symbolic and sub-symbolic aspects of vector-symbolic cognitive models. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 34, 2012.

[102] Ross W Gayler. Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience. *arXiv preprint cs/0412059*, 2004.

[103] Geoffrey E Hinton. Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, 46(1-2):47–75, 1990.

[104] Quora: How many particles are there in the universe? `https://www.quora.com/How-many-particles-are-there-in-the-universe/`.

[105] Gaurav Kumar and Pradeep Kumar Bhatia. A detailed review of feature extraction in image processing systems. In *2014 Fourth international conference on advanced computing & communication technologies*, pages 5–12. IEEE, 2014.

[106] Google code archive: word2vec:. `https://code.google.com/archive/p/word2vec/`. [Online; accessed 2020-09-29].

[107] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[108] DA Rachkovskij. Estimation of vectors similarity by their randomized binary projections. *Cybernetics and Systems Analysis*, 51(5):808–818, 2015.

[109] Pegasus DAX XML Schema. `https://pegasus.isi.edu/documentation/development/schemas.html`,.

[110] Pentti Kanerva et al. Fully distributed representation. *PAT*, 1(5):10000, 1997.

[111] Graham Bent, Patrick Dantressangle, David Vyvyan, Abbe Mowshowitz, and Valia Mitsou. A dynamic distributed federated database. In *Proc. 2nd Ann. Conf. International Technology Alliance*, 2008.

[112] Graham Bent, Patrick Dantressangle, Paul Stone, David Vyvyan, and Abbe Mowshowitz. Experimental evaluation of the performance and scalability of a dynamic distributed federated database. In *Proc. 3rd Ann. Conf. International Technology Alliance*, 2009.

[113] J. Ahrenholz, C. Danilov, T.R. Henderson, and J.H. Kim. Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7, Nov 2008. doi: 10.1109/MILCOM.2008.4753614.

[114] Workflow Generator Pegasus, . `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`.

[115] Graphviz - Graph Visualization Software. `http://www.graphviz.org/Home.php`.

[116] Joseph P Macker and Ian Taylor. Orchestration and analysis of decentralized workflows within heterogeneous networking infrastructures. *Future Generation Computer Systems*, 2017.

[117] Daniel Harborne, Chris Willis, Richard Tomsett, and Alun Preece. Integrating learning and reasoning services for explainable information fusion. *International Conference on Pattern Recognition and Artificial Intelligence*, 2018.

[118] Daniel Harborne, David Braines, Alun Preece, and Rafal Rzepka. Conversational control interface to facilitate situational understanding in a city surveillance setting. *The fourth Linguistic and Cognitive Approaches to Dialog Agents Workshop (LACATODA 2018)*, 2018.

[119] Semantic Bank. `https://flask.palletsprojects.com/en/2.0.x/`.

[120] etal N. Suri. The anglova tactical military scenario and experimentation environment. In *International Conference on Military Communications and Information Systems*. ICMCIS, 2018.

Chapter 10

Appendix

International Technology Alliance in Distributed Analytics & Information Sciences

Initial Program Plan

Applicable Period: Sept 21, 2016-January 15, 2018

Research Vision

With the explosion in low cost phones, wearables and the Internet of Things, most coalition operations will take place in an environment with a diverse set of small elements capable of computation, storage and communication. We propose leveraging the various devices available across the coalition members to create a system with distributed collaborative and cooperative capabilities. This interconnected system will provide an infrastructure for performing analytics required for coalition operations. It will leverage all the services offered by a wired backend infrastructure (e.g. a backend cloud system, data center or available cellular network infrastructure) but it will not be critically dependent on a continuous connectivity to the backend.

We envision a future where the interconnected system operates seamlessly across networks and systems belonging to different organizations (i.e. coalition members or sub-groups within a single coalition member). This system is frequently charged with performing tasks that require creating dynamic groups on a short notice. Such dynamic groups may be short-lived (days or hours), but could also last for a longer period (months). Differences in the pedigree of disparate systems belonging to different organizations necessitate the development of approaches that work with partial visibility, partial trust, and cultural differences, while simultaneously dealing with the challenges of a dynamically changing situation in which power, computation and connectivity may be severely constrained.

We want the ability to create an intelligent interconnected system, i.e. a system that can analyze the situation on the ground in real-time, anticipate the situation likely to happen in the future, and determine whether the situation requires human involvement. If the situation does not require human involvement, the system would undertake the most appropriate automatic action to the situation. When the situation needs human involvement, the system will recommend alternative courses of actions, along with their pros and cons. We refer to this capability that coordinates different elements, with opportunistic assistance from a fixed infrastructure with interrupted connectivity, as the *distributed coalition intelligence*.

2,5 and 10 Year Goals

The goal of our basic research is to discover and formulate the scientific principles that enable the physical realization of *distributed coalition intelligence* at the conclusion of our 10-year research agenda. This physical realization will require the transition of our basic research into the appropriate systems and solution development. We use the metaphor of a *distributed brain* to describe the end-vision. Just as the human brain is made of two parts, a left hemisphere and a right hemisphere, the distributed coalition intelligence will be an aggregation of several smaller sub-brains, each sub-brain belonging to a coalition member. All of the sub-brains work in a coordinated manner to perform analytics, and leverage the assets and knowledge available across the entire system. Just like the left hemisphere and right hemisphere of the human brain react differently to different stimuli, we expect different sub-brains to react differently in any situation, but the overall distributed system coordinates the different reactions in a seamless manner as needed. A pictorial representation of the concept is shown in Figure RP-1.

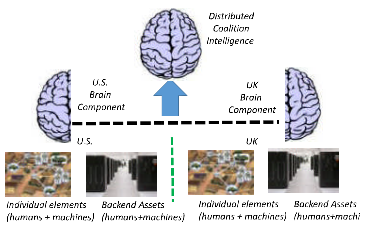


Figure RP-1

To attain the 10-year goal outlined above, we need to understand the fundamental principles underlying some of the key properties of the distributed coalition intelligence when applied to analytics. Our 5-year goal is to understand those principles underlying those properties.

In order to achieve our strategic vision, we must get an insight into the following properties by the end of 5-years.

- *Composability*: How do we compose smaller elements into a larger aggregate that works like a seamless whole? What are the principles that link the attributes of a component to the larger whole, and how can we compose components belonging to different organizations with partial visibility and control in an environment with limited resources?

- *Interactivity*: How do different computing elements and people interact with each other, both with other members of the groups and to external stimulus from the environment? How should we model and understand the interactions between different elements and information sources? How do different sub-brains work together as a larger aggregate brain under?
- *Optimality*: How can elements work together to obtain the optimal results in an environment with constrained resources? How can analytics be performed so that optimal performance is obtained automatically, instead of requiring complex manual optimization?
- *Autonomy*: How can elements work together in a proactive manner understanding future situations sufficiently well to operate with a degree of autonomous behavior? How can a system determine that autonomous operation is inappropriate and human intervention is needed? How can different elements simplify the cognitive burden involved to best assist humans in the loop when intervention is needed?

Understanding the principles behind these four attributes will allow us to attain significant capabilities for military defense as articulated in the UK MoD Technology Roadmap and in the U.S. third offset strategy.

Our six projects are defined so that the insights we obtain from them can be combined to help us understand the underpinnings of the four attributes. Our current view on how the different projects can be linked together to obtain the understanding of the four properties at the 5-year point is shown in Figure RP-2. Specifically, we plan on combining the results from projects P1, P2, P3 and P4 to understand the principles underlying composability, the results from projects P2, P4 and P6 to understand the principles that explain interactions among groups, the results from projects P1, P3 and P5 to understand how to self-optimize a system under limited resources, and the results from projects P2, P5 and P6 to understand the principles of autonomy. The results from the 5 years will be combined to get insights into principles governing the distributed coalition intelligence, which will enable us to physically realize such a system in 10 years by combining these scientific insights with appropriate systems building efforts.

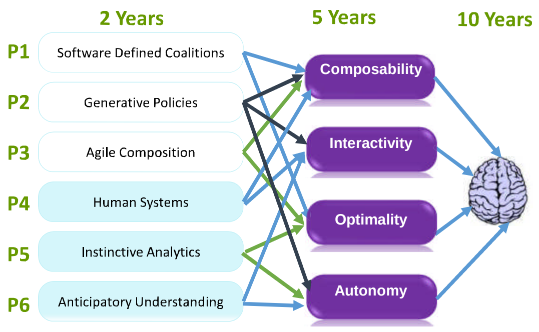


Figure RP-2

Project P5: Instinctive Analytics in a Coalition Environment

Project Champion: Thomas La Porta, Pennsylvania State University	
Primary Research Staff	**Collaborators**
Bongjun Ko (IBM-US)	Alun Preece (Cardiff)
Graham Bent (IBM-UK)	Flavio Bergamaschi (IBM-UK)
Heesung Kwon(ARL)	Geeth de Mel(IBM-UK)
Ian Taylor (Cardiff)	Ramya Raghavendra (IBM-US)
Jorge J Ortiz (IBM-US)	Swati Rallapalli (IBM-US)
Lance Kaplan (ARL)	
Leandros Tassiulas (Yale)	
Percy Liang (Stanford)	
Peter Waggett (IBM-UK)	
Sebastian Stein (Southampton)	
Thomas La Porta (PSU)	

The success of future military coalition operations—be they combat or humanitarian—will increasingly depend on the coalition's ability to share data and data processing services (e.g., aggregation, summarization, fusion) at the network edge. This is mainly because, future coalition networks will be composed of a set of heterogeneous assets—exposing their capabilities through micro-service architectures—that come together in an ad hoc manner to fulfill coalition needs, thus realizing a fully distributed information processing eco-system across the coalition boundaries.

However, one needs to have the means to bring these services together in a seamless and timely manner to support decision making in this new operational context; traditional approaches in which knowledge about services are centralized to match against user requirements will no longer scale in this setting. Thus, in this project we propose a system in which users specify their needs in a declarative manner, and the system infers required services (or compositions) by automatically discovering (or composing) them with respect to the declared user needs. We propose mechanisms in which services are discovered by features that are learnt over time so that the knowledge about these distributed services is derived in an (semi) automated manner. Furthermore, we will make our approach context sensitive (i.e., relevant to user needs) while being also sensitive to the uncertainties in the operating environment. Ultimately—with all these components together—we envision a brain-inspired computing paradigm to automatically match distributed coalition services to requirements so that the full potential of the future military networks may be realized.

We believe the analytic services available in a coalition environment should be automatically discovered and matched to support the tasks at hand without requiring any complex input from the user. We call such an infrastructure instinctive—an infrastructure that reacts automatically to address the analytics task at hand. In order to meet this objective, this project will undertake research into theorems, frameworks and mechanisms required to create instinctive analytics infrastructures. This will require bringing together multiple strands of research from the fields of machine learning, knowledge representation, optimization, and systems engineering[66],[67]. Within the

[66] Johnson, M. P., Rowaihy, H., Pizzocaro, D., Bar-Noy, A., Chalmers, S., La Porta, T. F., & Preece, A. (2010). Sensor-mission assignment in constrained environments. Parallel and Distributed Systems, IEEE Transactions on, 21(11), 1692-1705