Efficient Orchestration of Node-RED IoT Workflows Using a Vector Symbolic Architecture

Chris Simpkin*, Ian Taylor*, Daniel Harborne*,
Graham Bent†, Alun Preece*,
Raghu K. Ganti‡
*School of Computer Science and Informatics, Cardiff University
{simpkinc, taylorij1, HarborneD, PreeceAD }@cardiff.ac.uk
†IBM Research UK
{gbent}@uk.ibm.com
‡IBM Research USA
{rganti}@us.ibm.com

***Abstract*—Numerous workflow systems span multiple scientific domains and environments, and for the Internet of Things (IoT), Node-RED offers an attractive Web based user interface to execute IoT service-based workflows. However, like most workflow systems, it coordinates the workflow centrally, and cannot run within more transient environments where nodes are mobile. To address this gap, we show how Node-RED workflows can be migrated into a decentralized execution environment for operation on mobile ad-hoc networks, and we demonstrate this by converting a Node-RED based traffic congestion detection workflow to operate in a decentralized environment. The approach uses a Vector Symbolic Architecture (VSA) to dynamically convert Node-Red applications into a compact semantic vector representation that encodes the service interfaces and the workflow in which they are embedded. By extending existing services interfaces, with a simple cognitive layer that can interpret and exchange the vectors, we show how the required services can be dynamically discovered and interconnected into the required workflow in a completely decentralized manner. The resulting system provides a convenient environment where the Node-RED front-end graphical composition tool can be used to orchestrate decentralized workflows. In this paper, we further extend this work by introducing a new dynamic VSA vector compression scheme that compresses vectors for on-the-wire communication, thereby reducing communication bandwidth while maintaining the semantic information content. This algorithm utilizes the holographic properties of the symbolic vectors to perform compression taking into consideration the number of combined vectors along with similarity bounds that determine conflict with other encoded vectors used in the same context. The resulting savings make this approach extremely efficient for discovery in service based decentralized workflows.**

***Index Terms*—Decentralized Workflows, Vector Symbolic Architecture, Machine Learning, Dynamic Wireless Networks**

I. Introduction

This article is an extension of the work presented at WORKS 2018 [1] and describes a new technique that uses dynamic vector truncation to enable improved economies of bandwidth by exploiting the holographic nature of VSA representations.

Data analytic applications are increasingly making use of distributed software services that support the rapid construction of new applications by dynamically linking the services into different workflow configurations. New virtualization technologies (*e.g.,* containers, virtual machines) have fuelled this growth with a growing trend of constructing complex applications from smaller, repeatable components (*i.e.,* microservices). In addition, the emergence of smart devices and sensors, many of which are located at the edge of wireless networks, collectively known as The Internet of Things (IoT), represents a rapidly burgeoning requirement for distributed service workflows that can be rapidly reconfigured to perform a variety of distributed data analytic tasks. Increasingly IoT devices and services are being used in more challenging decentralized communication environments, such as Mobile Ad-Hoc Wireless Networks (MANETs) [2, 3, 4], where constructing and running applications that support distributed analytics introduces a much more diverse set of requirements [5]. In such environments, specifically, this work is targeting military environments such as the Internet of Battlefield Things (IoBT), a MANET is an interconnected set of wireless (and satellite) nodes that form a multi-hop network infrastructure which often results in low bandwidth, highly transient connectivity where endpoint stability, network bandwidth, and connectivity remain limited and transient and it becomes impractical, if not impossible, to support centralized service registries and to manage workflows executing at the edge.

Due to the complexity in controlling distributed service workflows, a Workflow Management System (WFMS) is often employed to assist in the data and task partitioning. A WFMS provides a robust means of describing applications, the control, data dependencies and the logical reasoning necessary for distributed execution. It is often used to automate processes

that are frequently executed or to formalize and standardize processes. Such workflows may be used to define and run computational experiments or to conduct recurrent processes on observational, experimental and simulation data.

Some WFMS use a programmatic style for data and task partitioning, for example, Swift/T [6] and the very successful Pegasus [7] scientific WFMS. Whereas others, such as Triana [8] and Node-red [9], use service-based workflows and include a visual programming environment to enable black box compute/data nodes to be wired together graphically.

For resource allocation and scheduling, modern WFMS typically rely on the high bandwidth, stable, networks to maintain a world view of the available resources from which it can employ complex resource allocation and scheduling algorithms to optimize the throughput of a shared set of compute 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. In the emerging IoT type environments, this type of WFMS becomes impractical and there is, therefore, a need for a new bandwidth-efficient WFMS approach that can enable distributed application construction and workflow orchestration without the need for a central point of control.

In previous work [1, 5, 10, 11, 12], we have addressed this challenge by making use of a Vector Symbolic Architecture (VSA) [13, 14, 15, 16] to encode functional representations of micro-services and workflows into symbolic semantic vector representations. In the previous implementations, we used 10,000 bit binary vectors to represent service descriptions and a hierarchical binding scheme to represent workflows. We described how VSA vectors can then be exchanged across a Mobile Ad-Hoc Network (MANET) using multicast, to perform service discovery, workflow construction and execution. Using this approach, micro-services can, using efficient vector matching operations, be discovered an organized into the required workflow in a completely decentralized way. The approach is ideally suited to the transient, low bandwidth, environments typically found in MANET environments.

In [1] we showed specifically how a VSA can be used to encode Node-Red IoT services and associated workflows and how the component services could be dynamically discovered and executed in a MANET without the need to specify IP locations and presented an implementation of this approach. In this paper, we augment our VSA approach to focus on bandwidth efficiency for service discovery. Our research contribution of this paper is to present a new dynamic VSA vector compression that is capable of applying a lossless compression scheme, which allows the vectors to be compressed for communication, without effecting the VSA bindings and comparison performance. In this scheme, we use the underlying VSA approach to bind services into Workflow vector representations and then we dynamically reduce (truncate) the size of the VSA binary vectors. The scheme leverages the fact that VSA vectors represent a true superposition of their sub-feature vectors, like a hologram, which enables partial vectors to be transmitted when the information content of the VSA vector being transmitted is low. We show that the size of the truncated vector depends on the number and semantic similarity of service vectors that are bound into the workflow vector. Further, we present a theoretical proof of the capacity of VSA vector superposition and show how this can be used to perform dynamic message sizing while maintaining the ability to discover and orchestrate complex workflows.

The rest of the paper is structured as follows. In the next section, we provide an overview of related work. In Section III, we describe how the VSA approach is used to encode service representations and in Section IV we describe how VSA enables workflows to be encoded and orchestrated. In Section V, we outline a simple use case that demonstrates how the VSA enabled Node-RED can perform decentralized data analytics. Section VI describes the architecture we have employed to enable existing services to participate in a distributed workflow. Section VII describes the implementation and methods used to meld our VSA architecture with Node-RED. In Section VIII, we then provide a description of our compression scheme, along with a mathematical proof and in Section IX we present some quantitative evaluation result that demonstrates how the compression scheme works using example workflows being constructed in an emulated MANET environment. Finally in Section X we draw conclusions and outline the scope of our future work.

II. Related work

For wired networks, there have been a wide variety of workflow systems developed [17, 18, 19, 20, 21, 22, 23, 24, 25, 26]. Many systems, such as Hadoop/mapReduce [27, 28] focus on delivering high-speed data analytics via highly parallel data processing techniques in high bandwidth communication environments on single cluster server farms. Geo-Distributed MapReduce, for example, G-Hadoop [29], attempts to implement Hadoop/MapReduce techniques across widely dispersed, heterogeneous data centres. Dolev et al. [30], surveys such attempts and concludes that geo-distributed big-data processing is highly dependent on task assignment, data locality, data movement, network bandwidth, security, and privacy. For IOT, with the advent of 4G and 5G networks, more and more research is focusing on moving the data analytics task to the edge. For example, Apache Edgent [31] focuses on data analytics at the edge, typically to exploit sensor data and perform anomaly detection at the edge.

We are not proposing that this VSA architecture might replace such systems because our VSA approach is tackling a problem that such systems do not need to solve; that is, the dynamic discovery of services without using centralized registries. For more conventional data analytics environments, 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. Whereas the target environment for our VSA architecture is low bandwidth, highly transient MANET networks such as those operating in military battlefield scenarios.

In such environments, 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 [5]. 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; such workflows are focused on operating in highly available distributed computing infrastructures using TCP, centralized management, and service discovery. Service-oriented systems, such as Taverna, have some support for discovery [32] but service providers are centralized and require manual configuration.

Consequently, service discovery is a key component in a transient distributed networked environment. Service discovery is a difficult problem even when services are hosted in centralized repositories, mainly because services are developed and deployed independently or developed by loosely cooperating developers in open environments. This has led to a complex mix of disparate service architectures employing different methodologies for the description of their inputs, outputs, and configurations. Even with standardized protocols, such as Multicast Domain Name Service (mDNS, [33, 34]) there are no conventions for service templates. In support of such situations, we are investigating vector based representations as a means of representing service descriptions that can be semantically compared within particular contexts, in an extremely resource-efficient way. Using such vectors, semantically rich queries in the form of vectors, can be sent out to the network, using protocols such as multicast for efficient querying in a complex space.

Hyperflow [26] is based on a formal model of computation called Process Networks, which uses asynchronous signals to coordinate flow. Such signals could operate in a decentralized way but currently, there is no service discovery component, rather it relies on the node.js [35] execution environment and employs third-party tools, such as *RabbitMQ* [36], to coordinate services. Petri net workflows [37] 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 [38] 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 only supports 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 [39] 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 [40] business workflow system (based on a high-level type of Petri nets) does support runtime discovery of the service objects required to execute a business workflow and is an excellent implementation for mainstream internet eCommerce 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.

In summary, like mainstream SOC, the fundamental challenges continue to be 1) Discovery and 2) Interoperability. However, to fulfil the time-critical operational goals of our target environment, the discovery of alternate workflow paths and services becomes a much more critical objective since the 'best' path or service may not be available or indeed part of it may go out of service during workflow execution!

In terms of 1) Discovery, we believe that the VSA approach offers advantages over the reference semantic Web service architecture because, in short, VSA, being a distributed representation, converts what would typically be a hierarchical XML service description based on some ontology into a superposition of sub-features, that is, it represents all of the category and value data of the XML description, 'simultaneously' as a single value. XML service descriptions that are similar to each other, regardless of order, create VSA vector values that are near to each other in the VSA vector space. Thus, we believe service matching and discovery is greatly simplified. Calculating the hamming-distance between two VSA service description vectors provides a simultaneous comparison of all service sub-features in a position-independent way and gives a graded match. In addition, semantic vector word embeddings such as those built using word2vec [41] can be leveraged to solve the ontology matching issues described in [42]. For example, the category 'LastName' is easily matched with 'Surname' and 'FamilyName'. (Note, it is straightforward to convert real number word embeddings into binary VSA embeddings that maintain the semantic relationships between words). In addition, using VSA to describe workflows objects as well as service objects will enable our VSA architecture to mine alternate workflow paths through the resulting VSA vector space, another future work objective.

With respect to 2) Interoperability, while we have not solidified a new approach to this complex problem, our current test

cases achieve inter-service communication by simply posting to known endpoints, the VSA architecture can support the current state of the art in this respect. This can be achieved during workflow discovery via the local arbitration step referred to in Section IV and fully described in [11, Section 7.2, Page 79]. To support the conversational negotiation required to connect service objects at discovery time the available protocols supported by a particular service object can be encoded as a sub-feature of the service's binary VSA description vector. In this way, when a VSA requester multicasts a request for a partner service, the responding services that best match will likely have a matching protocol. During the discovery process the local arbitration step, carried out by the requester, is then leveraged to negotiate a protocol and confirm that the selected partner is indeed able to communicate with the requester. Should this negotiation fail then the requester service can simply choose another responder or re-issue the query for another responder. In addition, we discuss in Section III-B and in [11, Section 4, Page 74] how the interoperability problem might be simplified using VSA because parameter positions and data types can be encoded in a simple manner. Further, using the role-filler pairs methodology, as described in Section III-B and [11], we see it as quite feasible to pass small data (numbers, strings, etc.) as parameters bound into in a VSA vector. Large data can be passed indirectly as a VSA vector by encoding a file name/ID, which is the method used for our Pegasus example. In the traffic congestion example, we encode the endpoint name directly in the VSA workflow for return to the Node-Red WMS. A future work objective is to consider how to pass all data as semantic VSA pointers since this would potentially neutralize the interoperability problem because, by definition, VSA vectors hold both the data value and are a description of the data value, see semantic pointers [43]. In addition, using VSA to describe workflows objects as well as service objects enables the VSA architecture to discover alternate workflow paths through the resulting VSA vector space.

VSA, a term originally coined by Gayler [14], use very large vectors to represent objects and features of objects within a hyper-dimensional vector space such that objects and concepts that are semantically similar to each other in the real world are positioned closer to each other in the vector space. VSAs can be based on real-valued vectors, such as in Plate's Holographic Reduced Representation (HRR) [13], or large binary vectors, such as Kanerva's Binary Spatter Codes (BSC) [15] that, typically, have $N \geq 10,000$. For this work, we have chosen to use Kanerva's BSCs but we note that most of the equations and operations discussed should also be compatible with HRRs [44].

Since VSA vectors are stochastic in nature [45] they are not compressible in the traditional sense. However, because VSA vectors represent data in a distributed manner, *i.e.,* each vector element participates in the representation of many sub-feature entities, and each sub-feature entity is represented collectively by many elements of the VSA vector [44], any reasonable length sub-section of a VSA vector can equally represent all of the original data. The decodability of such a sub-segment, so that all sub-features of the original vector can be decoded, is directly related to its dimensionality, *i.e.,* its length, or its number of bits. Plate, [13], gives a thorough mathematical analysis of the capacity of real number VSA vectors including a mathematical derivation for the capacity when vectors are similar [13, Appendix B.2]. Recchia/Kanerva et al. perform a computational efficiency analysis of binary VSA compared to real number HRR VSAs and obtain some empirical results for the capacity of binary VSA vectors in [46] and Kleyko derives a formula for the capacity of VSA vectors from which he calculates that the capacity of a 10kbit VSA vector to be approximately 89 [16]. The idea of truncating VSA vectors based on the number of sub-feature vectors contained does not seem to appear in the literature probably because VSA vectors have not been used as a basis for communication across networks before this work.

III. Encoding Service Representations using the Vector Symbolic Architecture

In [10] we describe in detail the use of VSAs to represent service workflows. This section provides a brief recap of the core principals to provide context for the Node-RED integration.

A. Vector Symbolic Architecture background

A common technique for achieving such semantic representations is to represent a high-level concept or feature by a collection of its sub-features in a hierarchically recursive manner[47] so that the sub-features themselves are also built up of their sub-features which in turn are built of their own and so on. Descending in this way, we eventually get down to a sub-feature that cannot sensibly be broken down further and define this as an *atomic* vector.

A key feature of VSA architectures is that all vectors have the same size; that is the vector for a high-level concept, such as the entire play Hamlet, is the same size as each of its sub-features, *i.e.,* acts, scenes, stanzas, sentences, words. In order to achieve this, sub-feature vectors are combined using a suitable bundling operator, which for BSCs[1] is *majority-vote* addition[10, 15]. The resultant vector is a *superposition* of all sub-features in the sum such that each vector element participates in the representation of many entities, and each entity is represented collectively by many elements [44]. Normalized hamming distance (HD) can be used to probe such a vector for its sub-features *without* unpacking or decoding the sub-features. *XOR* binding is used to build *roll-filler* pairs[13, 15] which allow sub-feature vectors to remain separate and identifiable (although hidden) within a concept vector superposition[10, 13, 15]. In addition, *binding* can be used to maintain positional and temporal relationships such as those needed for the execution of workflows.

Binding is commutative and distributive over superposition as well as being invertible [15, page 147]. This means that,

[1]Further references to operations used in VSA architectures are expressly talking about binary vector operations

if $Z = X \cdot A$ then $X \cdot Z = X \cdot (X \cdot A) = X \cdot X \cdot A = A$ since $X \cdot X = 0$, the zero vector[2]. 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:

$$Z = X \cdot A + Y \cdot B \tag{1}$$
$$X \cdot Z = X \cdot (X \cdot A + Y \cdot B) = X \cdot X \cdot A + X \cdot Y \cdot B \tag{2}$$
$$X \cdot Z = A + X \cdot Y \cdot B \tag{3}$$

Where $'.'$ indicates XOR binding and $'+'$ indicates *majority-vote-add*

Examination of Eq. (3) reveals that vector **A** has been exposed, thus, if we perform $HD(X \cdot Z, A)$ 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 features or symbols. XOR-*binding* also preserves distance, but produces a result that is uncorrelated to its operands. Hence, if $V = R \cdot A$ and $W = R \cdot B$ then $HD(V,W) = HD(A,B)$ even though, R, A and B have no similarity to V or W.

These operations allow us to create semantically comparable compound objects analogous to data structures as follows:

$$P1_v = FN_r \cdot John_v + SN_r \cdot Charles_v + Age_r \cdot 55yrs_v + Health_r \cdot T2Diabetic_v$$
$$P2_v = FN_r \cdot Lucy_v + SN_r \cdot Charles_v + Age_r \cdot 55yrs_v + Health_r \cdot T2Diabetic_v$$
$$P3_v = FN_r \cdot Charles_v + SN_r \cdot Smith_v + Age_r \cdot 55yrs_v + Health_r \cdot T2Diabetic_v$$

Note that without *role* vectors, *e.g.*, FN_r[3] then $HD(P1,P2) = HD(P1,P3) = HD(P2,P3)$ since each record would be an unordered bag of feature values. Thus *role* vectors can be used to perform the important function of categorization within a superposition. To test $P1$ for the surname $Charles_v$ we perform,

$$HD(xor(SN_r, P1_v), Charles_v) \tag{4}$$

For 10kbit vectors, if the result of Eq.(4) is less than 0.47 then the probability of $Charles_v$ being detected in error is less than 1 in 10^9 [15, page 143]. If our *person* records are 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 microservice, or node in a Parallel Distributed Processing network, having person records containing the surname $Charles_v$ and age $55years_v$ can check for a match and respond or become activated.

B. Encoding service descriptions into semantic vectors

As described in Section III-A a common approach for creating semantically rich representations is to represent a high-level concept as a collection if its sub-features in a recursive manner. Reviewing that X_r represents a role vector and Y_v a value vector, one such arrangement for services might be,

$$Z_v = Serv_r \cdot Serv_v + Resource_r \cdot ResP_v + QoS_r \cdot QoS_v \tag{5}$$
$$Serv_v = Inputs_r \cdot Inp_v + Name_r \cdot Name_v + Desc_r \cdot Desc_v + Outputs_r \tag{6}$$
$$Inp_v = One_r \cdot Float_r + Two_r \cdot Float_r + Three_r \cdot Float_r + One_r \cdot BitMap_r \tag{7}$$

[2]Throughout this text, unless otherwise stated $'.'$ indicates XOR binding and $'+'$ indicates *majority-vote-add*

[3]Note that throughout this text, a symbol having suffix r, (Y_r) represents a known *atomic*, *role* vector. A symbol having suffix v (X_v) indicates a vector that is representing a value.

Thus, Z_v, the high-level semantic vector representation of the service, is made up of a nested superposition of its sub-feature vectors. Listing 1 is an example of a JSON service description for one of the Node-RED object detectors in our Traffic Congestion use case. We now describe a new methodology for converting JSON service descriptions into a semantically comparable service vector descriptions.

Listing 1: 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 field-names within the JSON must be converted to unique role vectors and the JSON values must be converted to semantically comparable vector values. The value fields are encoded using Eq. (8) which is described fully in [10]. This means that values can be complex and are semantically comparable as long as, within a superposition, they are bound to the same roles.

When using field-names as roles to categorise the feature values of a service vector concept, one important issue is, how can we guarantee that the role vectors created are unique and have the same value across distributed service implementations. This is a particularly relevant question for Node-RED integration since Node-RED is open source and functional nodes/services can be created arbitrarily by an unrelated set of developers. In the original implementation we simply assigned, known, random hyper-dimensional vectors to each role/field-name, however, this does not allow for unrelated developers to invent new field-names and would require some sort of central database lookup so that distributed services agreed on the vector value of a role/field-name, otherwise they would not be able to perform semantic matching.

In this paper, we describe an alternate vector encoding method that ensures roles are always unique based on their, case insensitive, spelling. The encoding algorithm used for the field-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, since $XOR(A,\ A) = \mathbf{0}$ and $XOR(XOR(A,\ A),\ A) = A$. The algorithm to convert a field name to a vector is shown in Listing 2.

Listing 2: 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
```

To recursively encodes each feature chaining all field-names together with the sub-feature roll-filler pairs we use the *json_to_vecs*() function listed in Listing 3.

Listing 3: 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 field-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,
                chunkSentenceVector(str(json_input)).myvec
```

Where chunkSentenceVector creates semantically comparable vectors. 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 1 is shown in schematic form in Listing 4.

Listing 4: 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 4 represents a compound vector entry in the returned list. The rightmost vector is the value vector, all vectors to the left of this are unique role vectors. Each vector is XOR chained with the one to its left. Precedence is as follows:

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

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

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

Note, as mentioned above, that the output of *json_to_vecs*() is combined as a simple *majority_vote* bag of vectors, this helps makes the vectorization 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 [16].

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.

IV. Describing workflows using Vector Symbolic Architecture

A workflow is a set of inter-related tasks that must be carried out in a specific order. To compose a workflow some methodology is needed to describe the various steps and what data must be passed between each cooperating node in the workflow. In our previous work, we showed how Pegasus[48] DAX files could be parsed and converted into a VSA workflow. In the Pegasus implementation, such DAX files are written directly in XML script language, or, they can be generated programmatically via the Pegasus API, available in Java, Perl or Python. Node-RED provides a graphical means of describing a workflow by allowing graphical icons representing functional operations to have their input/outputs connected. Fig. 1 shows an outline of the Pegasus Montage_20 workflow composed via the Node-RED graphical interface.

In our previous work [10], we explained how we can combine functional vector service descriptions into a workflow via our hierarchical VSA binding scheme Eq. (8) and Eq. (9). The execution flow is achieved by sequentially unbinding using Eq. (10).

$$Z_x = \sum_{i=1}^{x} Z_i^i . \prod_{j=0}^{i-1} p_j^0 + StopVec . \prod_{j=0}^{i} p_j^0 \tag{8}$$

Omitting $StopVec$ for readability, this expands to,

$$Z_x = p_0^0 . Z_1^1 + p_0^0 . p_1^0 . Z_2^2 + p_0^0 . p_1^0 . p_2^0 . Z_3^3 + ... \tag{9}$$

$$Z'_{n+1} = (p_n^{-n} . Z'_n)^{-1} \tag{10}$$

Fig. 1: Pegasus Montage_20 composed using Node-RED.

In all of the above equations, the Z_n terms are the semantic vector representations built using the methods described in Section III-B. In addition, for very large workflows the Z_n term may be a cleanup vector representing a large grouping of smaller steps, or in Node-RED terms, analogous to a sub-flow.

In [10] we also explain how discovery and workflow orchestration can be achieved using the above equations. For a linear workflow, the workflow steps are bound and unbound using Eq. (8) and Eq. (10) respectively. The $p_0, p_1, p_2, \ldots$ vectors are role vectors used to define the current position/step in the workflow. After the workflow has been built the unbinding procedure essentially exposes each microservice description in turn. Flow is controlled by the currently active node doing its functional work and then performing the next unbinding using Eq. (10) to activate the next node, no central controller is needed.

Note that, because we are using semantic vector descriptions for each exposed vector service request we fully expect to get multiple replies. To avoid race conditions and to enable on the fly load balancing we employ a method of *local arbitration* described in [10, 11] whereby the currently active node acts as the local arbiter for selection of the next workflow step.

$$Z'_1 = (p_0^0(T + Z_x))^{-1} = p_0^{-1}.T^{-1} + Z_1^0 + p_1^{-1}.Z_2^1 + p_1^{-1}.p_2^{-1}.Z_3^2 + \ldots \quad (11)$$

$$Z'_2 = (p_1^{-1}.Z'_1)^{-1} = p_1^{-1}.p_0^{-2}.T^{-2} + p_1^{-1}.Z_1^{-1} + Z_2^0 + p_2^{-2}.Z_3^1 + \ldots \quad (12)$$

Equations (11) and (12) show the state of the workflow vector after the first and second unbinding. Only Z_1 is visible in Eq. (11) and Z_2 is visible in Eq. (12) because all other vectors are permuted by position vectors[10].

For DAG workflows we extend this mechanism by employing three phases:[10]

1) A recruitment phase where services are discovered, selected and uniquely named.
2) A connection phase where the selected services connect themselves together using the newly generated names.
3) An atomic *start* command indicates to the connected services that the workflow is fully composed and can be started.

Thus, in mathematical terms, using Eq. (9):

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

$$Recruit_{Nodes} = p0^0.Z_1^1 + p0^0.p1^0.Z_1^2 + \ldots p0^0.p1^0.p2^0.p3^0.Z_1^4 \ldots$$

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

The resulting workflow, WP, is a single vector superposition representing the linear sequence of steps needed to discover, connect and initiate the workflow. That is, the WP vector completely encapsulates all three phases needed to discover, connect and start the DAG workflow. Once injected into the network, via a multicast transmission, the vector will activate matching distributed nodes which, in turn, unbind and pass the vector around in a peer-to-peer fashion with no central point of control. Once the last connection is made and the DAG is composed and fully connected the arbitrary node that makes this final connection causes the workflow to start when it unbinds WP exposing the *Start* vector and multicasts this to the network.

During the Recruitment phase; *(a)* real services respond to matches via their VSA cognitive layer; *(b)* the currently active node uses local arbitration[10] to select the best node for the next Recruit nodes step; *(c)* selected nodes build representations of their *parent* and *child* vectors which will be used during the Connection phase so that each service can be informed of its inputs and outputs. Notice that, during the Connection phase, the WP vector has become a sequential list of alternating parent $\mathbb{P}$ and child $\mathbb{C}$ vectors. This is how each recruited node learns of its partner connections. Control continues to pass from node to node but, during the connection phase, when a node becomes activated by seeing its *parent* vector it simply unbinds/multicasts the next vector, since in doing so it will activate its associated *child* service, automatically informing the child service of its IP address. When a service receives a multicast that matches its *child* vector it stores the parent's IP-address and multicasts a response informing the parent of its IP-address before unbinding and multicasting the next vector.

When the final child request is processed, this is detected by the $Connect_{Nodes}$ cleanup service[10] causing it to unbind and multicast the *StartVec* indicating to all nodes that the workflow has been fully constructed and processing can be started. At this point each *VSA-Agent* sends an **/init/** message to its associated *Workflow-Agent* and the proper work is initiated, see Section VII-A.

The scheme supports encoding of DAG workflows having

one-to-many, many-to-many, and many-to-one connections. In [10] we show that the result provides several desirable features and byproducts: it can encode workflows /sub-workflows that can be unbound on-the-fly and executed in a completely decentralized way; associated metadata can also be embedded into the vector, *e.g.,* security, configuration.; the vector representation is extremely compact and self-contained and can be passed around using standard group transport protocols; and semantic comparisons or searches are scoped within a sub-group of services in a workflow, allowing scoped service matchmaking.

V. Traffic Congestion Use Case

In prior work, we identified traffic monitoring as a plausible use case involving sensing (*e.g.,* via a network of traffic cameras) and decision making (*e.g.,* routing traffic to avoid congested areas) supported by an interactive question-answering interface ([49], [50]). The concept for this interface is to provide decision support for a user tasked with managing the state of city or region-wide traffic. In [49], we explored detecting traffic congestion using a number of services which could be both distributed and owned by multiple agencies (*e.g.,* operating as a coalition). In [50], we explored how natural language queries relating to traffic could be answered by taking advantage of the output of distributed data sources and processing services. In both these pieces of work, we did not outline the co-ordination of these distributed resources, merely providing specific architectures and the Node-RED workflows that could provide the required answers.

In this and the following sections, we show how VSA enabled Node-RED can be used to semantically describe and cognitively wrap the existing services and how we construct the workflow vector that is used to orchestrate the discovery and execution of the workflow across the distributed resources.

A. Data Sources & Processing Resources

For this work, the example we use (counting the number of cars on a given street) takes advantage of a subsection of data sources and resources used in prior work but our solution featured in this paper can be applied to working with a wider range of available services.

The main data source we have taken advantage of is the Transport for London (TFL) traffic camera API[4]. This allows access to imagery and video from around one-thousand traffic cameras situated around London. The imagery and video is updated every five minutes and the video provided is a ten second clip recorded at the beginning of the five minute interval.

To detect cars, we process the imagery from the traffic camera feeds using an object detector (MobileNetSSD[5]) supplied within the OpenCV (Open Source Computer Vision) library [6]. Finally, to convert the list of detected cars to a count we use a simple service that is designed to count the items in a list it receives. Having this as a service (and not hard coded in to the interface's processing of the result for example) allows for this "list to count" function to be used within workflow construction.

B. Moving to a Dynamic and Decentralized Environment

Within a decentralized environment, these resources need to be discovered dynamically amongst a distributed array of services. Once the nature of the query is established, the correct services must be identified and chained together to answer the query. During the discovery process there are two key considerations, services may be replicated identically providing redundancy and thus there may be multiple services that provide a "perfect fit" or the required functionality. These must be discovered and selected appropriately. Secondly there may be services available which, although do not meet the functionality exactly, still provide the functionality required. For example, when counting the number of vehicles on a road, a vehicle detector (a detector that identifies cars, bikes, vans etc) is a "perfect fit" but if detectors for these individual concepts exist (individual car detector, van detector etc), their output could be aggregated and provide an output that may still be appropriate if no vehicle detector is available.

A method of discovery and execution within a distributed setting must factor these two properties in to best take advantage of the resources made available and to maximize the queries that are answerable.

VI. Architecture

To manage and fuse these sensor feeds, an architecture is required that integrates the services in a loosely coupled way to support decentralized discovery and execution. This loosely coupled nature ensures that existing services and resources can be quickly set up to be discovered and take part in query responses without having to be re-written from the ground up.

In Fig. 2, we illustrate the three layers of architecture. The lowest layer (in grey), contains the services and resources we wish to make available. These can be existing or newly created, and can be unique services or redundant replications of the same service. Simply these are end points which can be sent a request and respond in kind.

The second layer (in green), contains our proposed solution to handling workflow execution, the workflow agents. These decentralized wrapper services are light weight and encompass the "real" services below them. They manage the address of the end point, the collection of the required input data, the retrieval of the end point's response and finally the forwarding of this response to the next workflow agent in the chain.

The highest layer (in orange), is where the VSA agents reside which, in our solution, handle the discovery of appropriate services using vector representations of the task (as detailed in section IV).They have the required information about the location of the workflow agent and a representation of the function the linked service provides.

In summary, within this architecture diagram, the vertically aligned agents and services represent the connected VSA

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

[5]MobileNet-SSD: https://github.com/chuanqi305/MobileNet-SSD

[6]https://github.com/opencv/opencv

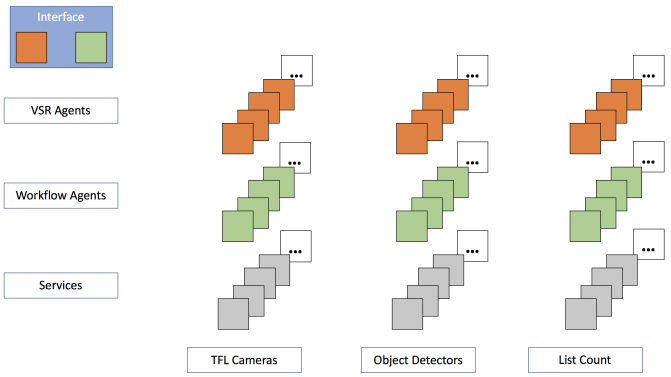

Fig. 2: Distributed architecture for answering the question "How many cars on Oxford Street?"

agent, the workflow agent and the service itself which work together to offer discoverability and execution of a particular function amongst the array of services. The distinct columns (three in our diagram) represent categories of service *i.e.,* the different functions that can be provided. The depth shown at each layer within these columns represents redundant or similar services for a function.

The VSA agent and the workflow agent are co-located on the same hardware but the service itself can be located on another platform that the workflow agent can communicate with. In our use case, for example, the camera service is a remote web-service that is only reachable from the node running the corresponding workflow agent. To participate in the VSA discovery scheme each 'real' service starts an instance of the VSA cognitive layer and associated flask wrapper service. We have trailed various methods of communication between the VSA cognitive layer and the real service. In the flask wrapper method, a simple flask wrapper is used as an interface between the 'real' service and the VSA layer. The existing service passes a description of itself, either in JSON, XML or DAX description language to the VSA cognitive component via the flask wrapper service. The VSA cognitive component builds a VSA vector that represents its associated service, opens a multicast listener and commences listening for discovery request on this vector. In this way, it enables discovery of the real service. Since the VSA layer is running on the same device as the flask wrapper, which is acting as a proxy to the real service, it could communicate to the wrapper via any number of standard IPC methodologies. As mentioned above our current implementation uses known endpoints. The wrapper is intended to be a lightweight bespoke interface to the real service and as such must be coded to enable communication with the original service. Section VII describes this in more detail and [11, Section 6] gives full details of the current communication stack.

VII. Node-RED integration

In Fig. 1, we illustrated a typical Node-RED workflow. To illustrate the integration of Node-RED with VSA, we make use of the linear workflow shown in Fig. 3 that is used in the simple traffic congestion use case.

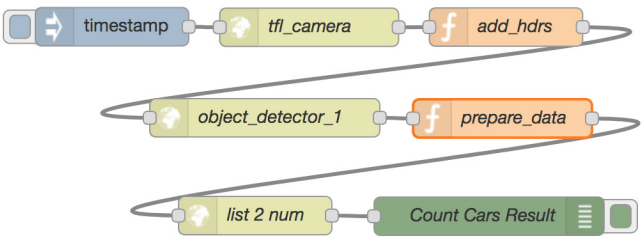

Fig. 3: Typical Node-RED workflow.

In a conventional Node-RED implementation all messages travel through the Node-RED workflow engine. This requires the location of external services to be specified and these must be known in advance. In this example, the external service, *object_detector*_1, is defined using an Node-RED http-request node as shown in Fig. 4.

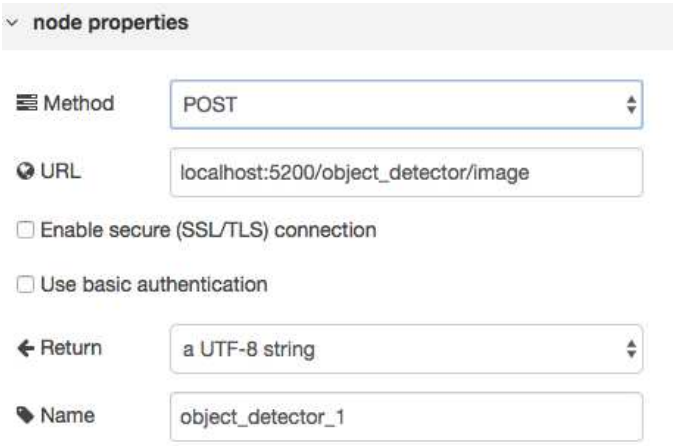

Fig. 4: HTTP Requester properties for Object_Detector_1

The HTTP request is enacted via the NODE-Red workflow engine. The Node-RED engine makes a POST to the address shown, it collects the reply and passes it, as a Node-RED payload message, to the next node in the flow. This means that all messages must pass through the central Node-RED controller and, since the external service endpoint is hard-coded, no alternate service can be selected. The architecture described in Section VI is largely independent of Node-RED. However, we integrate with Node-RED in two ways. First, Node-RED is used as a front end graphical composition interface, acting as the interface component located in the top left corner of Fig. 2. Second, we have extended the VSA Importer toolkit to parse Node-RED workflows, so we can export Node-RED workflows into a decentralized discovery and execution environment.

To integrate Node-RED as the front end to our VSA workflow architecture we implemented a new Node-RED

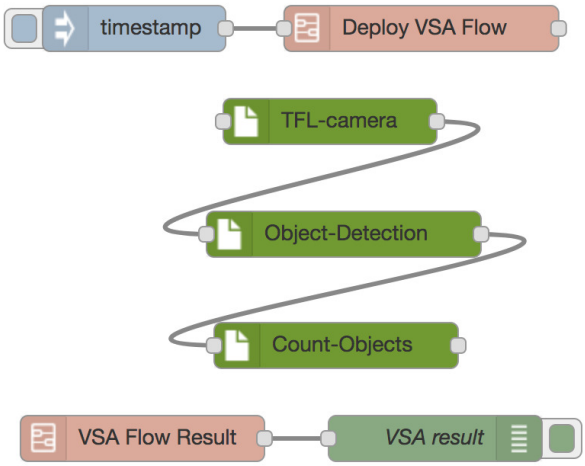

Fig. 5: VSA workflow composition using vsa_service node.

Fig. 6: Node-RED vsa_service properties.

node type, the *'vsa_service'* node. This node type has its message passing component disabled because message passing between distributed components is carried out by our VSA architecture. The *vsa_service* node has two properties, see Fig. 6. The 'Name' property is a standard Node-RED property. The 'JSON' property is used as an entry field to accept either a file name or a literal JSON string which is used to describe the service attributes and features of the particular service it is representing. This JSON description is encoded by the VSA architecture in to a single 10Kbit semantic vector as described in Section III-B. When passed as a filename, if the filename type is *.bin*, then the VSA architecture will load a previously built 10kbit vector during workflow encoding, otherwise it reads and vectorizes the JSON on the fly.

Real microservices that are participating in our VSA architecture also encode their functional description using the same methods, hence, when the Node-RED service request is deployed, the service flow, via the VSA architecture, can discover and utilize the real services. Message passing does not rely on returning each payload to the Node-RED engine. Rather, the VSA architecture performs the discovery, selection and connection of real worker services that are listening for work in a distributed network. Once the workflow nodes have been discovered and recruited and connected the VSA *Workflow-Agents* execute the workflow as described in Section VII-B, all messages are passed directly between the *Workflow-Agents* without a central point of control. When each terminal node *Workflow-Agent*, defined as one having no *child* endpoints, has completed its work, it returns its output, if any, to Node-RED via a HTTP POST.

In Fig. 5, the top right node, *'Deploy VSA Flow'* is a conventional Node-RED sub-flow that extracts the Node-RED flow JSON description from the flow's page and sends it to the VSA Workflow Importer using a HTTP POST request. A simplified listing of the JSON extracted by *'Deploy VSA Flow'* is shown below. The *wires:[]* list field, *id:* field, and *JSON:* field are used by the *VSA Workflow Importer* to build the workflow vector.

The bottom right node, *'VSA Flow Result'* is a conventional Node-RED sub-flow that implements a Node-RED *'HTTP in'* node having endpoint **/vsa_work_done/**. This endpoint is used by the *Workflow-Agents* to return their results to Node-RED.

Listing 5: Node-RED flow JSON.

```
{ "nodes": [
    { "wires": [
        ["695e3ae2.e16854"]
      ],
      "name": "tfl-camera",
      "JSON": "tfl_camera_tenyson_road.json",
      "y": 295,
      "x": 495,
      "z": "1592cb0c.f2c005",
      "type": "vsa_service",
      "id": "226dec54.165fe4"
    },
    { "wires": [
        ["dee64892.9d1a28"]
      ],
      "name": "Object-Detection",
      "JSON": "object_detection.json",
      "y": 295,
      "x": 702,
      "z": "1592cb0c.f2c005",
      "type": "vsa_service",
      "id": "695e3ae2.e16854"
    },
    { "wires": [
        []
      ],
      "name": "Count-Objects",
      "JSON": "count_objects.json",
      "y": 295,
      "x": 915,
      "z": "1592cb0c.f2c005",
      "type": "vsa_service",
      "id": "dee64892.9d1a28"
    }
  ],
  "id": "1592cb0c.f2c005",
  "label": "VSA Car Counter"
}
# Node-RED JSON listing with non-vsa_service nodes removed.
```

A. *Implementation*

We simulate real world operation using CORE/EMANE [51], a real-time network emulator. From Fig. 2, each *VSA-Agent* and *Workflow-Agent* are co-located and run in

their own VM, each of which has its own IP-Address on a simulated wireless mesh network. We instantiate multiple similar services in separate VMs so that we simulate having multiple possible services capable of satisfying a particular workflow step. Node-RED runs on the host ubuntu server, thus, we simulate an extended distributed/MANET environment for our services and request flows from Node-RED. Between and during runs we can move services in and out of range taking them in and out of service.

Our VSA platform is implemented in `Python2` and has a modular architecture with several components that are capable of being reused as plugins to other systems:

1) **CORE/EMANE** All VSA and Workflow agents are started by loading a CORE configuration file defining each of our services. Each service starts in its own VM. the *VSA-Agent* loads its semantic vector service description and starts listening on the VSA multicast address for semantic vector messages.
2) **The Workflow Importer** component uses a general plugin infrastructure that allows VSA to parse multiple formats. It has an implementation for the Pegasus workflow description(DAX) and for this new implementation, we added a module for parsing Node-RED workflows. Once parsed the resulting graph is formed using VSA primitives: *NodeVectors* and *EdgeVectors* for further processing by the VSA Creator.
3) **The VSA Creator** is used to bind the lists of vectors into a single vector, a reduced representation, of the workflow using Eq. (8) and chunking[10, page 3]. Chunking is performed bottom-up and vectors are recursively rebound until the vector list (workflow) is reduced to a single vector. The *NodeVectors* list and the *EdgeVectors* list are combined separately producing two high level vectors, the $Recruit_{Nodes}$ vector and the $Connect_{Nodes}$ vector. The VSA Creator then binds these two vectors together with the *Start* vector into a single vector representing the entire workflow, the *WorkFlow* vector. This *WorkFlow* vector and all its associated sub-vectors are encapsulated in a *chunk tree* object[10, page 3] which is then passed to the VSA executor.
4) **The VSA Executor** implements the *Workflow Agents* part of Fig. 2 by providing a decentralized overlay for wrapping the underlying services. The services themselves can be conventional request/response *e.g.,* Web/REST interfaces, and the role of the Workflow Agents are to bind to these underlying services and wiring the inputs and outputs to such services to serve the service in a decentralized way. This local wrapping aspect of the implementation is important because it enables decentralization over non-decentralized services. The VSA Executor essentially *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 decode the first vector in a higher level vector. For robustness, the VSA Executor can be made to distribute more than one copy of the cleanup objects into other terminal node objects.
5) **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.
6) **The Visualization Component** takes the log output and generates a DAG layout graph using Graphviz [52].

B. *Workflow Execution*

The Workflow-Agents (**WA**) are currently implemented as python flask services. The VSA component knows the endpoint of the **WA** which is wrapping the underlying functional service. The **WA** has a number of HTTP endpoints/routes that are used to control it and facilitate message passing between nodes.

1) **/init/** The *VSA-Agent* POSTs the list of its discovered input/outputs to its partner *Workflow-Agent*. The *parent* addresses are used as keys in a python tracker dictionary to collect data on this **WA's** inputs.

```
{
"name": "tfl_camera", "server_id": "192.168.0.72:4612"),
"child_connections": [['192.168.0.72', 4612], ['192.168.0.72', 4614]],
"parent_connections": [['192.168.0.72', 4623], ['192.168.0.72', 4617]]
}
# /init/ input message from VSA-Agent


{
"192.168.0.72:4623": "False",
"192.168.0.72:4617": "False"
}
# Input tracker Dictionary
```

2) **/start/** Those services that do not have any input(*parent*) connections call their DoWork() function and send the resulting data to each **/work/** endpoint in this worker's *child* list. Those services that do have parent connections return and await ***/work/*** messages. All messages are currently passed as a JSON dictionary with the following format, (Note that the sender's *listening address:port* is used for the *server_id* field because it uniquely identifies the sender.)

```
{
"name": "tfl_camera" # The Sender's name string
"server_id": "192.168.0.72:4612" # The Sender's server address
"data": "A valid JSON serializable python object"
"status": "good" # Alternatively "UNEXPECTED"
}
# Workflow_Agent: Work data message.
# Input and output messages have the same format.
```

3) **/work/** On receipt of a 'work' message each work message is stored in the input tracker until all inputs have been received. At this point, the DoWork() function is called passing in the data from the messages it received. Any output from the DoWork() function is then sent to each **/work/** endpoint in this worker's *child* list. If the Workflow-Agent receives an empty *child* connections list via the **/init/** message it is considered a terminal node and POSTS its output, if any, back to the known Node-RED listener.

4) **DoWork()** The DoWork function is specific to each task and must be customized by the developer who is implementing, or wrapping, a real service. For a producer service, *e.g.,* a tfl_camera, it simply packages and returns its data. For a producer-consumer, it processes the data collected and returns a packaged response which will usually be some transformation of its input data.

Further details of the VSA platform and workflow execution can be found in [11].

VIII. A Mathematical Model for VSA Vector Truncation Optimization

In Section III-A we explained that the symbolic vectors we use are typically 10,000 bit vectors. In previous work, we have maintained the vector size and have exchanged vectors of this size irrespective of the number of bound sub-vectors that it contains. One of the important properties of these large binary vectors is that they are a distributed or holographic representation of the bound sub-vectors. As such, if the number of sub vectors is small then successful comparisons can still be made if both vectors are truncated to the first n-bits. This holographic property of the symbolic vectors suggests an approach for compressing the message payloads that are exchanged over the communications network. Essentially vectors can be truncated without effecting the VSA bindings and comparison performance. In this section, we consider the mathematical basis for such a scheme and later, in Section IX, 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. 10, or an un-ordered bag of roll-filler pairs as described in Section III-B describing the functionality of a micro-service.

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*, or unordered such as a *person_record*. In VSA, these type of compound vectors are commonly called *chunk* vectors and the number of sub-feature vectors in a chunk is its *ChunkSize* [46, 53].

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 binding via Eq. 8 the permutation vectors ensure that each sub-feature vector is orthogonal to the other vectors in the sum.

Another important property is that for VSA vectors *majority_vote* addition creates a *superposition* of the sub-vectors such that each sub-vector is represented by many, but not all, binary bits of the chunk vector and each sub-vector has a unique random distribution of bits in the chunk vector. Hence, the chunk vector could be considered a digital analogue of a hologram in that each sub-vector is represented equally in any reasonably long sub-segment of the chunk vector. For example, the expected hamming distance of a chunk vector S containing two sub-vectors $S = [A+B]$ is $HD = 0.25$. This means that 75% of the bits in S will match to A and a different 75% of the bits in S will match with B. If we compare S to a random vector not contained S then it is easy to see that 50% of the bits will match. Thus, 25% of the bits in A, (or B) actively differentiate it from a random vector. If we take any reasonable length segment of S, for example, the first 1000 bits and perform a hamming distance comparison to A we will still get approximately the same 0.25 result. Hence, if there are only a small number of sub-vectors in a chunk then we can truncate the chunk vector before transmission and any listening VSA-Service agent will still be able to match to the vector. On *'seeing'* a truncated vector the VSA-Service simply truncates its vector description to the same length and performs a hamming distance match.

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 contained. The degree of similarity between sub-vectors in the known vocabulary/cleanup memory is also an important factor in determining the minimum size to which a chunk vector can be truncated. When matching to a truncated chunk vector, it is necessary to be able to distinguish between the actual sub-vector embedded in preference to some similar sub-vector that is not part of the chunk. Truncating a chunk vector to its minimum useful size is a direct corollary for the capacity of a chunk vector which has been extensively studied [13, 16, 53]. Since VSA vectors are effectively an IID sequence of 1s and 0s with probability 0.5 they can be effectively modelled via the binomial distribution as shown in Fig. 7.

For a VSA vector of dimension D, Eq. 13 shows the expected value (normalized hamming distance) μ_y of a single sub-vector when it is compared to a chunk vector containing n sub-vectors.

$$\mu_y = 1 - \frac{1}{2^m} \sum_{i=\lfloor m/2 \rfloor}^{m} \binom{m}{i} \quad \begin{cases} m = n & if\ n\ even. \\ m = n-1 & if\ n\ odd. \end{cases} \tag{13}$$

where

$$\binom{m}{i} \tag{14}$$

is the number of combinations of i from m. The corresponding variances are given by:

$$\begin{aligned} \sigma_y &= \sqrt{\mu_y * (1-\mu_y)/D} \\ \sigma_x &= \sqrt{\mu_x * (1-\mu_x)/D} \end{aligned} \tag{15}$$

The combined probability distribution is:

$$N(\mu_x - \mu_y,\ \sigma_x^2 + \sigma_y^2) \tag{16}$$

The probability of decoding one sub-vector successfully is given by:

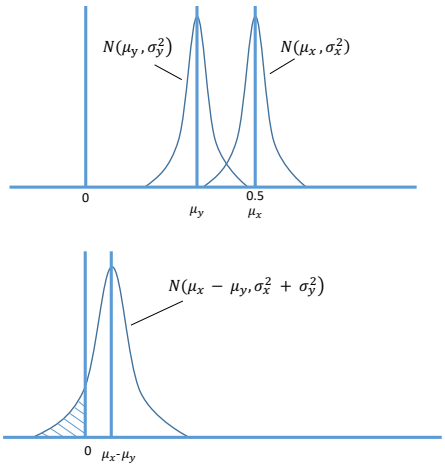

Fig. 7: Vector hamming distance distributions for random vectors with mean $\mu_x = 0.5$ and a vector comprising m sub-vectors having a mean μ_y. Also shown is the resulting combined probability distribution.

$$P_1 = \int_{\infty}^{0} N(\mu_x - \mu_y,\ \sigma_x^2 + \sigma_y^2) = 0.5\left(1 + erf\left(\frac{-(\mu_x - \mu_y)}{\sqrt{2}\left(\sigma_x^2 + \sigma_y^2\right)^{1/2}}\right)\right) \tag{17}$$

and similarly probability of decoding all m vectors is given by:

$$P_m = (1 - P_1)^{m(m-1)} \tag{18}$$

Using Eq. 13 through Eq. 17 it is possible to compute the minimum number of bits that are required to ensure that the unbound target service 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 between the target service and other services. This is shown in Fig. 8.

The different curves show the minimum number of bits needed for various levels of similarity between the target service vector and other service vectors. The greatest compression is obtained when all services are orthogonal to each other.

When a potentially matching cognitively enabled service is attempting to make a comparison between the unbound vector and its vector it needs to use an appropriate hamming distance threshold to determine if it does indeed match. The threshold value is again obtained from Using Eq. 13 through Eq. 17. The HD-Expected curve is the expected hamming distance whilst the HD-Upper Bound is the required threshold value, taking account of variance, to ensure that a match is only possible with an error of 1 in 10e6. This upper-bound occurs when the compression method assumes zero similarity between listening VSA-Services because, referring to Fig 8, the highest similarity is obtained when expected similarity between alternate services is zero.

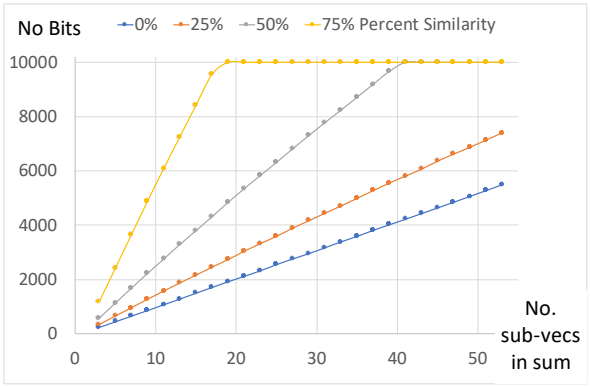

Fig. 8: Minimum message size (No Bits) for chunk vectors containing n sub-feature vectors at differing *similarity factors*.

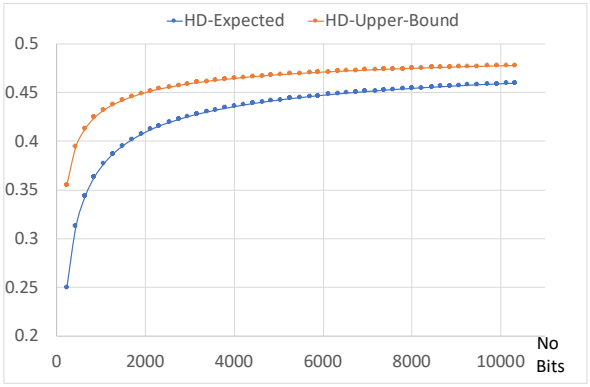

Fig. 9: Required threshold to ensure correct match as a function of the vector size (No of bits)

In the following section we show how the theoretical model is used in practice.

IX. Application of Theoretical Model In Practice

The mathematical model presented in Section VIII can be applied to complex workflows, which involves multiple levels of symbols that are combined (bound) together to represent higher concepts (services and workflows). In practice, the binding strategy, along with the data representation, ultimately has an impact on the level we can compress (truncate) the vectors whilst maintaining the comparable integrity of the vectors we wish to differentiate between. To highlight this, consider the extreme case in which we are required to differentiate between two vectors that are only different in one, unknown, bit position. Clearly, no compression would be possible. In other words, the number of symbols we choose to combine along with commonality, (or *similarity factor*)

of those combined vectors between different higher level concepts defines the probabilistic bounds on the the amount we can truncate the vectors during transmission on the network. As a simplified example, consider the case where we want to compare workflow vectors built from a limited set of defined symbols:

1) $SINE + FFT + POWER\_SPECTRUM$
2) $SINE + FFT + AUTO\_CORRELATION$
3) $AUDIO\_STREAM + AUTO\_CORRELATION + EVENT\_DETECTION$
4) $SINE + EVENT\_DETECTION + DEEP\_LEARNING$

In the above simple example, comparing at the workflow level, workflow 1 has a 66% similarity to any other workflow in the network. This is the same for Workflow 2. Workflow 3, due to ordering (recall that the binding process contains a permutation, see Eq. (8), to establish ordering), has 0% correlation with other workflows. Workflow 4 has 33% correlation with other workflows in the system. It is intuitive to see, in this case, that Workflow 3 should have the best level of compression, followed by Workflow 4, then Workflows 1 and 2 would have the worst rate of compression. This concept is used within the overall strategy, which can be described with the aid of Fig. 8 and Fig. 9.

We applied these ideas to our original test-cases which were fully described in [10]. All service descriptions (service objects) and workflows graphs are still built with 10kbit VSA vectors as described in III-B and workflow objects are built using Eq. (8). The discovery and orchestration scheme remains unchanged as described in Section III-B, except for the inclusion of the minimum vector size calculation and truncation of the vectors before transmission.

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

The interest here was to investigate how much bandwidth could be saved by truncating the vector requests for both

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

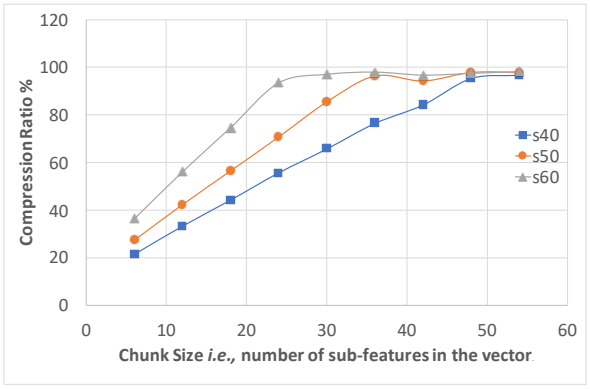

Fig. 10: Minimum message size (No Bits) for chunk vectors containing n sub-feature vectors.

discovery and orchestration, on the fly, based on the number of sub-vectors that each VSA concept vector contains. 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 Fig. 8 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% the Fig. 8 indicates that the 10kbit vector can be truncated to 1680 bits. This vector is unbound to reveal the Act_1 vector and multicast to the network. Each service compares this vector with the first 1680 bits of their vector and measure the hamming distance. Since the number of bits is now 1680 the service computes the threshold from the information in Fig. 9, 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. If it responds and is selected then it takes its clean vector which is 5 scenes long and requiring ten vectors so it truncates the vector to 2751 bits and the process continues.

Table I shows the results obtained 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 I, column *'10k'* is the bandwidth consumed without vector truncation. Columns *'s60'* and *'s50'* are the 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.

Tables II and III 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 and so these runs were conducted at similarity factors of *'s50'* and *'s40'*. Note that, the Pegasus workflow examples were used as a means to test that the VSA architecture could successfully encode, discover and connect DAG workflows in a verifiable way. The savings listed represent purely the savings in the message bandwidth generated for discovery and parameter passing. We do not suggest that it would be sensible to attempt to execute these highly data intensive tasks in our low bandwidth transient environment. No actual data processing was executed since our CORE/EMANE network emulator does not have enough compute power. The Pegasus runs were used to verify that such DAG workflows could be discovered and connected without a central point of control and had more hardware been available these workflow tasks would have been executable.

Fig. 10 shows how compression ratio varies with chunk size and similarity factor and is an experimental verification of the theoretical compression ratios shown in Fig. 8. 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 Fig. 10 is that small chunk sizes give better compression ratios and it should be noted that the recursive nature of Eq. (8) enables the use of small chunk sizes for concepts containing many sub-features, however, in [11] we showed that smaller chunk sizes reduce the semantic matching capabilities of the resulting concept vectors. This is an area for future investigation.

X. Conclusions And Future Work

In this paper we have identified that the majority of existing workflows rely on centralized management and therefore require a stable endpoint to deploy such a manager. One such workflow system is Node-RED, which is designed to bring workflow-based programming to the IoT. However, the majority of scientific workflow systems, and specifically systems like Node-RED, are designed to operate in a fixed networked environment, which rely on a central point of coordination to manage the workflow.

In more dynamic settings, such as MANETs, on demand workflows that are capable of spontaneously discovering multiple distributed services without central control are essential. In these types of environments 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 which operates in a decentralized manner.

We have described how to migrate Node-RED workflows into a decentralized execution environment, so that such workflows can run on Edge networks, where nodes are extremely transient in nature. We have demonstrated that such a new class of workflow can be realized by using vector symbolic architectures (VSA) in which symbolic vectors can be used to encode workflows containing multiple coordinated sub-workflows in a way that allows the workflow logic to be unbound on-the-fly and executed in a completely decentralized way.

We have demonstrated the feasibility of such an approach by showing how we can migrate a centralized Node-RED based traffic congestion workflow into a decentralized workflow by adding a cognitive-aware wrapper which uses the VSA to semantically represent the component services and the associated workflow. The traffic congestion algorithm is implemented as a set of Web services within Node-RED and we have architected and implemented a system that proxies the centralized Node-RED services using cognitively-aware wrapper services, designed to operate in a decentralized environment.

We further extend this work by introducing a new dynamic VSA vector compression scheme that compresses vectors for on-the-wire communication, thereby reducing communication bandwidth while maintaining the semantic information content. This algorithm utilizes the holographic properties of the symbolic vectors to perform compression taking into consideration the number of combined vectors along with similarity bounds that determine conflict with other encoded vectors used in the same context. From the test-case results we note that, while the resulting bandwidth savings may appear low in terms of MB saved, and would be not 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 unmanned aerial vehicles or even soldiers move around, distancing themselves from their nearest neighbour with increasingly less bandwidth and even becoming fragmented from the network.

Symbolic vector representation can also be used to represent not just the workflow but also the semantics of the component services at various levels of semantic abstraction. This leads

directly to the concept of self-describing services and data. We believe that in future the VSA approach offers the potential to combine the workflow, self-describing services and data into vector representations that will enable alternative service compositions to be automatically constructed and orchestrated to perform tasks specified at higher levels of semantic description. Our future work will therefore focus on such self-describing service compositions in order to realize the vision set out in [5].

XI. Acknowledgements

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.

References

[1] C. Simpkin, I. Taylor, D. Harborne, G. Bent, A. Preece, and R. 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)*. IEEE, 2018, pp. 52–63.

[2] S. Corson and J. Macker, "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations," RFC 2501 (Informational), Internet Engineering Task Force, Jan. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2501.txt

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

[4] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva, "A performance comparison of multi-hop wireless ad hoc network routing protocols," in *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*. New York, NY, USA: ACM, 1998, pp. 85–97.

[5] D. Verma, G. Bent, and I. Taylor, "Towards a distributed federated brain architecture using cognitive iot devices," in *9th International Conference on Advanced Cognitive Technologies and Applications (COGNITIVE 17)*, 2017.

[6] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[7] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman, "Workflow Management in GriPhyN," in *Grid Resource Management*, ser. International Series in Operations Research & Management Science, J. Nabrzyski, J. Schopf, and J. Weglarz, Eds. Kluwer Academic Publishers, Dordrecht, 2003, vol. 64.

[8] I. Taylor, M. Shields, I. Wang, and O. Rana, "Triana Applications within Grid Computing and Peer to Peer Environments," *Journal of Grid Computing*, vol. 1, no. 2, pp. 199–217, 2003.

[9] "Node-RED: Flow-based programming for the Internet of Things," https://nodered.org/.

[10] C. Simpkin, I. Taylor, G. A. Bent, G. de Mel, and R. 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*. IARIA, 2018, pp. 21–27.

[11] C. Simpkin, I. Taylor, G. A. Bent, G. de Mel, S. Rallapalli, L. Ma, and M. Srivatsa, "Constructing distributed time-critical applications using cognitive enabled services," *Future Generation Computer Systems*, vol. 100, pp. 70–85, 2019.

[12] C. Simpkin, I. Taylor, G. A. Bent, G. De Mel, and S. 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.

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

[14] R. W. Gayler, "Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience," *arXiv preprint cs/0412059*, 2004.

[15] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors." *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/cogcom/cogcom1.html#Kanerva09

[16] D. Kleyko, "Pattern recognition with vector symbolic architectures," Ph.D. dissertation, Luleå tekniska universitet, 2016.

[17] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of scientific workflows in the askalon grid environment." *SIGMOD Record*, vol. 34, no. 3, pp. 56–62, 2005.

[18] 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*. Springer, New York, 2007, ch. ASKALON: A Development and Grid Computing Environment for Scientific Workflows, pp. 143–166.

[19] 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)*. IEEE Computer Society, New York, 2004, pp. 423–424.

[20] P. Kacsuk, "P-grade portal family for grid

infrastructures," *Concurr. Comput. : Pract. Exper.*, vol. 23, pp. 235–245, March 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1654

[21] 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. Katz, "Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems," *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, 2005.

[22] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, November 2004.

[23] A. Harrison, I. Taylor, I. Wang, and M. Shields, "WS-RF Workflow in Triana," *International Journal of High Performance Computing Applications*, vol. 22, no. 3, pp. 268–283, Aug. 2008. [Online]. Available: http://hpc.sagepub.com/cgi/doi/10.1177/1094342007086226

[24] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan, "The trident scientific workflow workbench," in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 317–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=1488725.1488936

[25] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, "Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR," *Int. J. High Perform. Comput. Appl.*, vol. 22, pp. 347–360, August 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1400050.1400057

[26] B. Balis, "Increasing scientific workflow programming productivity with hyperflow," in *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*, ser. WORKS '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 59–69. [Online]. Available: http://dx.doi.org/10.1109/WORKS.2014.10

[27] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[28] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[29] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen, "G-hadoop: Mapreduce across distributed data centers for data-intensive computing," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 739–750, 2013.

[30] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer, "A survey on geographically distributed big-data processing using mapreduce," *IEEE Transactions on Big Data*, vol. 5, no. 1, pp. 60–80, 2017.

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

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

[33] M. Giordano, "DNS-Based discovery system in service oriented programming," *Lecture notes in computer science*, vol. 3470, p. 840, 2005.

[34] D. Steinberg and S. Cheshire, *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[35] L. M. Surhone, M. T. Tennoe, and S. F. Henssonow, *Node.Js*. Mauritius: Betascript Publishing, 2010.

[36] "RabbitMQ is the most widely deployed open source message broker," https://www.rabbitmq.com/.

[37] W. M. P. van der Aalst, "The application of petri nets to workflow management." *Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[38] J. P. Macker and I. Taylor, "Orchestration and analysis of decentralized workflows within heterogeneous networking infrastructures," *Future Generation Computer Systems*, vol. 75, pp. 388 – 401, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17300262

[39] W. Shakespeare, *The Tragedy of Hamlet*. University Press, 1904.

[40] J. Fabra, P. Álvarez, J. A. Bañares, and J. Ezpeleta, "Deneb: a platform for the development and execution of interoperable dynamic web processes," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 18, pp. 2421–2451, 2011.

[41] Y. Goldberg and O. Levy, "word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.

[42] B. T. Le, R. Dieng-Kuntz, and F. Gandon, "On ontology matching problems," *ICEIS (4)*, pp. 236–243, 2004.

[43] C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen, "A large-scale model of the functioning brain," *Science*, vol. 338, no. 6111, pp. 1202–1205, Nov. 2012. [Online]. Available: http://www.sciencemag.org/content/338/6111/1202

[44] T. A. Plate, *Holographic Reduced Representation: Distributed Representation for Cognitive Structures*. Stanford, CA, USA: CSLI Publications, 2003.

[45] M. Sahlgren, A. Holst, and P. Kanerva, "Permutations as a means to encode order in word space," 2008.

[46] G. Recchia, M. Sahlgren, P. Kanerva, and M. N. Jones, "Encoding sequential information in semantic space models: comparing holographic reduced representation and random permutation," *Computational intelligence and neuroscience*, vol. 2015, p. 58, 2015.

[47] G. E. Hinton, "Mapping part-whole hierarchies into connectionist networks," *Artificial Intelligence*, vol. 46, no. 1-2, pp. 47–75, 1990.

[48] "Workflow Generator Pegasus," https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.

[49] D. Harborne, C. Willis, R. Tomsett, and A. Preece, "Integrating learning and reasoning services for explainable information fusion," *International Conference on Pattern*

Recognition and Artificial Intelligence, 2018.

[50] D. Harborne, D. Braines, A. Preece, and R. 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.

[51] J. Ahrenholz, C. Danilov, T. Henderson, and J. Kim, "Core: A real-time network emulator," in *Military Communications Conference, 2008. MILCOM 2008. IEEE*, Nov 2008, pp. 1–7.

[52] "Graphviz - Graph Visualization Software," http://www.graphviz.org/Home.php.

[53] P. Kanerva *et al.*, "Fully distributed representation," *PAT*, vol. 1, no. 5, p. 10000, 1997.