Dynamic Distributed 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*—There are a large number of workflow systems designed to work in various scientific domains, including support for the Internet of Things (IoT). One such workflow system is Node-RED, which is designed to bring workflow-based programming to 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 in order to manage the workflow. The main focus of the work described in this paper is to investigate means whereby we can 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. In this work, we demonstrate the feasibility of such an approach by showing how we can migrate a Node-RED based traffic congestion workflow into a decentralized environment. 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. Our cognitive services use a Vector Symbolic Architecture to semantically represent service descriptions and workflows in a way that can be unraveled on the fly without any central point of control. The VSA-based system is capable of parsing Node-RED workflows and migrating them to a decentralized environment for execution; providing a way to use Node-RED as a front-end graphical composition tool for decentralized workflows.**

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

I. Introduction

During the last decade, there has been an explosion in the quantity, variety and complexity of data generated routinely by research and industry. This has been driven by the development of new virtualization technologies (e.g. containers, virtual machines, and so forth) which allow applications to adapt elastically on-demand, as well as the adoption of service interfaces (e.g. micro-services) which break down complex problems into smaller and repeatable tasks. 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 communications and data analytics in a distributed environment.

A scientific workflow is a set of interrelated computational and data-handling tasks designed to achieve a specific goal. The workflow methodology provides a robust means of describing applications consisting of control and data dependencies along with the logical reasoning necessary for distributed execution. It is often used to automate processes which 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.

The majority of scientific workflow management systems utilise a centralised control methodology in order to simplify the complex task of distributing and load balancing workflow steps across, often heterogeneous, compute resources. In addition, while IoT devices are widely distributed geographically, the current approach for management of such devices is cloud based and therefore similarly centralised. However, considering the huge and growing volume represented by the IoT it will become more and more expensive and impractical to manage and coordinate billions of devices in centralised server farms.

Further, more and more compute resource is becoming available at the edge of networks in the form of traditional mobile devices as well as IoT devices. Hence an opportunity is emerging to utilise such edge resources for data analytics and a need is evident for an alternate decentrailised approach to managing IoT devices.

Vector Symbolic Architectures (VSAs) [1, 2, 3, 4] are a set of lossy dimensionality reduction methodologies that enable large volumes of data to be compressed into a fixed size vector in a way that captures associations and similarities as well as enabling categorizations between data to be built up. Such vector representations are recursive as originally proposed by Hinton [5] in that they allow for higher level abstractions to be formulated in the same format as their lower level components. VSAs are capable of supporting a large range of cognitive tasks such as; *(a)* Semantic composition and matching; *(b)* Representing meaning and order; *(c)* Analogical mapping; *(d)* Logical reasoning; *(e)* They are highly resilient to noise; *(f)* They have neurologically plausible analogues which may be exploited in future distributed cognitive architectures. Consequentially they have been used in natural language processing [6, 7, 8] and cognitive modeling [9, 10].

In this paper we describe the use of a VSA to architect a mechanism that can be used for distributed discovery and orchestration of remote devices and compute resources without any knowlege of the 'IP' location of such devices and without the need for a central point of control. We then show how it can be used to implement a new operational mode for Node-RED [11], a well known cloud based, graphical workflow management system for IoT devices. We describe the use of this new mode to transition an existing Node-Red, traffic congestion, workflow to operate in CORE/EMANE [12], a real-time network emulator. Further, we describe how the VSA can be used to orchestrate more complex data analytics tasks and, as a test case, use the new Node-Red mode to run a distributed simulation of the Montage Pegasus[13] workflow in the CORE/EMANE environment.

Our scheme converts the existing Node-Red microservices into a cooperating set of decentralized proxies, which are instantiated into the CORE environment, by adding a cognitively-aware wrapper around each service to facilitate decentralized discovery and execution. Request workflows are then composed using the Node-Red graphical interface to describe the connectivity and functional requirements of each workflow step but without specifying IP locations as per the normal Node-Red scheme. The user interface is then used to launched the workflow request into the CORE environment for distributed discovery and execution.

The VSA cognitive wrapper represents individual services as symbolic vectors which are in turn bound into compound vectors in order to represent complex workflow structures [14]. This scheme is used to represent workflows by combining services, edges, sub-workflows, branches, etc. into a hierarchical set of vectors that represent the workflow as a whole. Thus, this extremely compact representation can be transmitted to multiple services (e.g., using multicast) and only services that semantically match a particular vector will proceed with the work; hence they are cognitive. The vectors have the property that they can then be unbound to unravel the workflow and therefore, the decentralized operation is simplified to a transmit-unbind and re-transmit procedure, since each unbind unravels the next step of the wortkflow.

The simple, and yet powerful approach can provide semantically rich representations of workflows that have several interesting by-products, including: *(a)* the ability to make semantic comparisons at each level of the architecture (e.g., semantic searches are scoped within a sub-group of services in a workflow); and *(b)* and the ability to bind extra metadata along with the workflow structure, which is only accessible to the services that can consume it (e.g. it enables policy enforcement).

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 decentralised 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. Finally in Section VIII 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 [15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. On the other hand, on-demand distributed analytics workflows for general collaborative environments need spontaneous discovery of multiple distributed services without central control [25]. 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 more focused on operating on highly available distributed computing infrastructures using TCP, using centralized management and service discovery. Service-oriented systems, such a Taverna, have some support for discovery [26] 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 input, outputs, and configurations. Even with standardized protocols, such as Multicast Domain Name Service (mDNS, [27, 28]) 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 [24] is based on a formal model of computation called Process Networks, which uses asynchronous signals to coordinate flow. Such signals could operate in an decentralized way but currently, there is no service discovery component, rather the node.js [29] execution environment employs the use of third party tools eg. RabbitMq to coordinate services. Petri net workflows [30] 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 [31] 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, etc. 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 [32] 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 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.

III. Encoding Service Representations using the Vector Symbolic Architecture

In [14] 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. Vector Symbolic Architectures 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.

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[5] 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 which cannot sensibly be broken down further, and define this as an *atomic* vector. VSAs can be based on real valued vectors, such as in Plate's Holographic Reduced Representation (HRR) [1], or large binary vectors, such as Pentti Kanerva's Binary Spatter Codes (BSC) [3] that, typically, have $N \geq 10,000$. For this work, we have chosen to use Kanerva's BSCs but we note that most of equations and operations discussed should also be compatible with HRRs [9].

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[3, 14]. 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 [9]. 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[1, 3] which allow sub-feature vectors to remain separate and identifiable (although hidden) within a concept vector superposition[1, 3, 14]. In addition *binding* can be used to maintain positional and temporal relationships such as those needed for execution of workflows.

Binding is commutative and distributive over superposition as well as being invertible [3, page 147]. This means that, 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 has 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

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

[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.

to perform the important function of categorisation 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 [3, 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,

$$Z_v = Serv_r \cdot Serv_v + Resource_r \cdot ResP_v + QoS_r \cdot QoS_v \tag{5}$$

$$Serv_v = Inputs_r \cdot Inp_v + Name_r \cdot Name_v + Desc_r \cdot Desc_v \;\; + Outputs_r \tag{6}$$

$$Inp_v = One_r \cdot Float_r + Two_r \cdot Float_r + Three_r \cdot Float_r + One_r \cdot BitMap_r \tag{7}$$

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 (8) which is described fully in [14]. 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 below.

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 right most vector is the value vector, all vectors to the left of this are unique role vectors. Each individual 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 vectorisation of JSON service descriptions immune to ordering issues but does limit the number of service line entries to approximately 100, the maximum capacity of a single 10kbit binary vector [4].

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. In 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[13] DAX files could be parsed and converted into a VSA workflow. Such DAX files can be written directly in XML script language, or, they can be generated programmatically via the Peagsus 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. Figure 1 shows an outline of the Pegasus Montage_20 workflow composed via the Node-RED graphical interface.

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

In our previous work [14] we explained how we can combine functional vector service descriptions into a workflow via our hierarchical VSA binding scheme (8) and (9). The execution flow is achieved by sequentially unbinding using (10).

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

Omitting $StopVec$ for readability, this expands to,

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

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

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 [14] 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 and (10) respectively. The $p_0, p_1, p_2, ...$ 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 (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. In order to avoid race conditions and to enable on the fly load balancing we employ a method of *local arbitration* described in [14] whereby the currently active node acts as the local arbiter for selection of the next workflow step.

$$Z_1' = (T + p_0^0.Z_0)^{-1} = p_0^{-1}.T^{-1} + \boxed{Z_1^0} + p_1^{-1}.Z_2^1 + p_1^{-1}.p_2^{-1}.Z_3^2 + ... \tag{11}$$

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

Equations (11) and (12) show the state of the workflow vector after the first and second unbinding. Only z_1 is visible in (11) and z_2 is visible in (12) because all other vectors are permuted by position vectors[14].

For Directed Acyclic Graph (DAG) workflows we extend this mechanism by employing three phases:[14]

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 (9):

$$\begin{aligned} WP \; = & \; p0^0.(Recruit_{Nodes})^1 + \\ & p0^0.p1^0.(Connect_{Nodes})^2 + \; p0^0.p1^0.p2^0.Start^3 \end{aligned}$$

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

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

The resulting workflow, WP, is a single vector superposition representing the linear sequence of steps needed to discover, connect and initiate the workflow. During the Recruitment phase; *(a)* real services respond to matches via their VSA cognitive layer; *(b)* the currently active node uses local arbitration[14] to select the best node for the next Recruit nodes step; *(c)* selected nodes build representations of their own *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 unbind/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 to its *child* vector it stores the parent's ip-address and multicasts a response informing the parent of its own 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[14] 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 [14] 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, etc.; 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 ([33], [34]). 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 [33], we explored detecting traffic congestion using a number of services which could be both distributed and owned by multiple agencies (i.e., operating as a coalition). In [34], 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

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

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 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 interfaces 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 in order 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 in order to best take advantage of the resources made available and to maximize the queries that are answerable.

VI. ARCHITECTURE

In order 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 be quickly set up to be discovered and take part in query responses without having to be re-written from the ground up.

In Figure 2, we illustrate the three layers of architecture. The lowest layer (in gray), 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

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

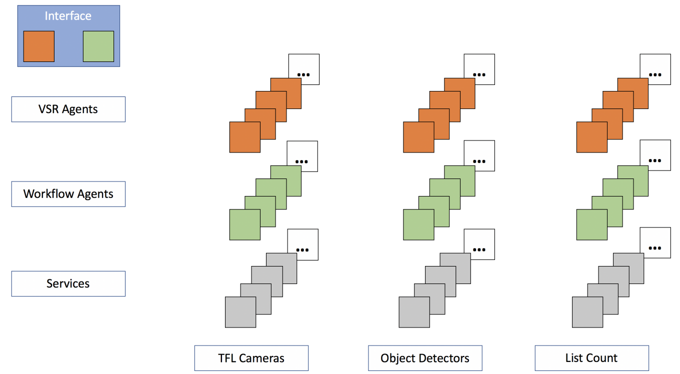

Fig. 2: Distributed architecture for answering the question ”How many cars on Oxford Street?”

of the end points 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 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 webs-service that is only reachable from the node running the corresponding workflow agent.

VII. NODE-RED INTEGRATION

In Figure 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 Figure 3 that is used in the simple traffic congestion use case.

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 Figure 4.

The HTTP request is actually 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

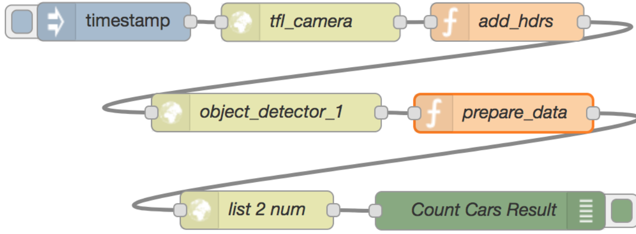

Fig. 3: Typical Node-RED workflow.

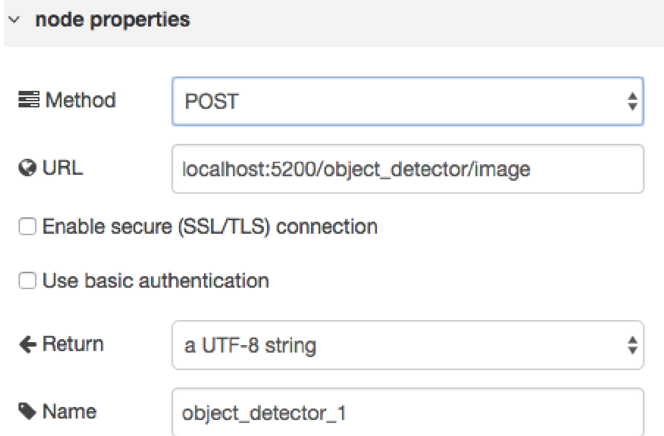

Fig. 4: HTTP Requester properties for Object_Detector_1

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 Figure 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.

In order to integrate Node-RED as the front end to our VSA workflow architecture we implement a new Node-RED node type, the *'vsa_service'* node. This node type has its message passing component disabled because message passing between distributed components its carried out by our VSA architecture. The *vsa_service* node has two properties, see Figure 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 vectorises the JSON on the fly.

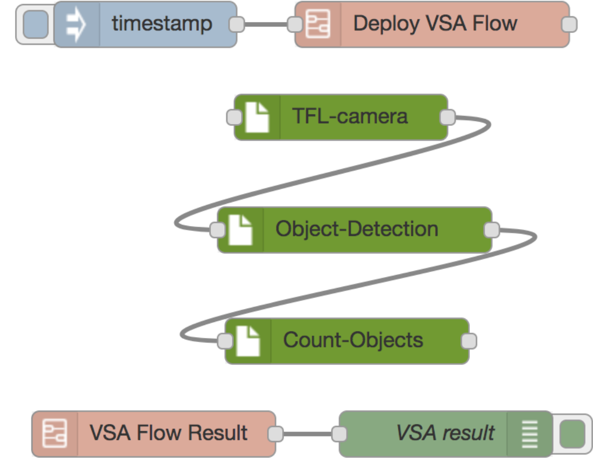

Fig. 5: VSA workflow composition using vsa_service node.

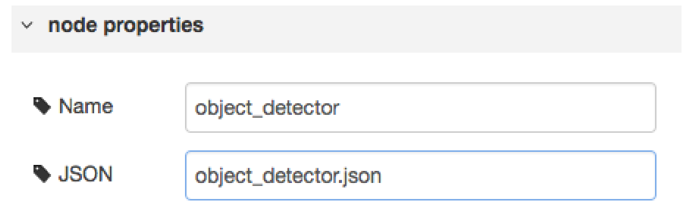

Fig. 6: Node-RED vsa_service properties.

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 utilise 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 Figure 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.

```
{ "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[12], a real-time network emulator. From Figure 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 (8) and chunking[14, 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[14, page 3] which is then passed to the VSA executor.
4) **The VSA Executor** implements the *Workflow Agents* part of Figure 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 Visualisation Component** takes the log output and generates a DAG layout graph using Graphviz [35].

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 used as keys in a python tracker dictionary for the purpose of collecting 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 serialisable 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 customised 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.

VIII. 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 in order 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 specificlly systems like Node-RED, are designed to operate in a fixed networked environment, which rely on a central point of coordination in order 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 we can 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.

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 [25].

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

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

[3] 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

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

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

[6] M. N. Jones and D. J. K. Mewhort, "Representing word meaning and order information in a composite holographic lexicon," *psychological Review*, vol. 114, no. 1, pp. 1–37, 2007.

[7] G. E. Cox, G. Kachergis, G. Recchia, and M. N. Jones, "Toward a scalable holographic word-form representation," *Behavior research methods*, vol. 43, no. 3, pp. 602–615, 2011.

[8] 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.

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

[10] 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

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

[12] 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.

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

[14] C. Simpkin, I. Taylor, G. A. Bent, G. de Mel, and R. K. Ganti, "A scalable vector symbolic architecture approach for decentralized workflows."

[15] 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.

[16] 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.

[17] 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.

[18] 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

[19] 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.

[20] 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.

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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.

[26] "Taverna Service Discovery Plugin," http:

//dev.mygrid.org.uk/wiki/display/developer/Tutorial+-+Service+discovery+plugin.

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

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

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

[30] 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.

[31] 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

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

[33] 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.

[34] 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.

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