Future Generation Computer Systems 100 (2019) 70–85

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Constructing distributed time-critical applications using cognitive enabled services

Chris Simpkin [a,*], Ian Taylor [a], Graham A. Bent [b], Geeth de Mel [b], Swati Rallapalli [c], Liang Ma [c], Mudhakar Srivatsa [c]

[a] *School of Computer Science and Information, Cardiff University, UK*
[b] *IBM Research, UK*
[c] *IBM Research, USA*

HIGHLIGHTS

- An Architecture for decentralized construction and control of time critical applications.
- Cognitively enabling existing services using a Vector Symbolic Architecture (VSA).
- Complex service workflow orchestration at the network edge with no central point of control.
- Semantic encoding of services, workflows, and the time-critical constraints for QoS and QoE.
- Empirical proof that VSA encoding methods are scalable to complex workflows.

ARTICLE INFO

Article history:
Received 14 May 2018
Received in revised form 15 March 2019
Accepted 3 April 2019
Available online 16 May 2019

Keywords:
Decentralized workflows
Quality of service
Quality of experience
Vector symbolic architectures
Distributed fitness functions
Time-critical applications
Dynamic wireless networks

ABSTRACT

Time-critical analytics applications are increasingly making use of distributed service interfaces (e.g., micro-services) that support the rapid construction of new applications by dynamically linking the services into different workflow configurations. Traditional service-based applications, in fixed networks, are typically constructed and managed centrally and assume stable service endpoints and adequate network connectivity. Constructing and maintaining such applications in dynamic heterogeneous wireless networked environments, where limited bandwidth and transient connectivity are commonplace, presents significant challenges and makes centralized application construction and management impossible. In this paper we present an architecture which is capable of providing an adaptable and resilient method for on-demand decentralized construction and management of complex time-critical applications in such environments. The approach uses a Vector Symbolic Architecture (VSA) to compactly represent an application as a single semantic vector that encodes the service interfaces, workflow, and the time-critical constraints required. 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 in a completely decentralized manner. We demonstrate the viability of this approach by using a VSA to encode various time-critical data analytics workflows. We show that these vectors can be used to dynamically construct and run applications using services that are distributed across an emulated Mobile Ad-Hoc Wireless Network (MANET). Scalability is demonstrated via an empirical evaluation.

1. Introduction

Time-critical analytics applications are increasingly making use of distributed service interfaces (e.g., micro-services) that support the rapid construction of new applications by dynamically linking the services into different workflow configurations. In traditional Service Oriented Architectures (SOA), operating over fixed networks, the underlying TCP/IP backbone guarantees sufficiently stable service endpoints and connectivity to facilitate the construction and management of multi-service applications using centralized management schemes. Using such schemes also supports the dominant approach to service discovery and matching, which is based on the use of centralized service registries to provide a catalogue of services, using formal ontologies to facilitate service definition and service matching. Such approaches, while often very effective, incur overhead in terms of the knowledge engineering effort required to create them [1]. For time-critical applications, centralized control also allows multiple workflow tasks to be load-balanced, scaled, and optimized across

* Corresponding author.
E-mail address: simpkinc@cardiff.ac.uk (C. Simpkin).

https://doi.org/10.1016/j.future.2019.04.010
0167-739X/© 2019 Elsevier B.V. All rights reserved.

multiple heterogeneous compute resources so that their Quality of Service (QoS) and Quality of Experience (QoE) requirements can be fulfilled.

In decentralized environments, such as Mobile Ad Hoc Wireless Networks (MANETs) [2–4], constructing and running applications that support group-oriented collaborative applications (e.g., multi-user chats) or distributed analytics [5,6], introduces a much more diverse set of requirements. In such environments end point stability 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. A similar requirement is emerging for the Internet of Things (IOT) where although more compute resource and potentially useful services are available at the edge of networks, both on traditional mobile devices and emerging IoT devices, these cannot be utilized because they cannot easily be connected together without some centralized, usually cloud based, control. Time-critical applications operating in such environments introduce further complexity requiring a capability for applications to rapidly reconfigure themselves in the event of change, so that their QoS/QoE requirements can be satisfied. There is therefore a need for new methods that can enable application construction and workflow orchestration without the need for a central point of control.

Micro-services are a variant of the SOA architectural style that structures an application as a collection of loosely coupled services that can be linked in different workflow configurations. They overcome the complexity of ontology-based service composition by having much simpler interfaces (usually RESTful interfaces) which enables these services to be developed by multiple parties without rigid standardization of service description templates. However, the lack of a formal template means that a mechanism is required that will allow semantic matching/discovery of the appropriate micro-services required for a given application. This mechanism needs to be *compact*, in order to minimize bandwidth requirements, and *flexible*, in order to minimize the knowledge engineering overhead required to generate the service descriptions. In this paper we describe a method by which these objectives can be achieved using the capabilities of Vector Symbolic Architecture (VSA) representations.

Vector Symbolic Architectures (VSAs) [7–10] are a family of bio-inspired methods for representing and manipulating concepts and their meanings in a high-dimensional vector space. They are a form of distributed representation that enables large volumes of data to be compressed into a fixed size feature vector in a way that captures associations and similarities as well as enabling semantic relationships between data to be built up. Such vector representations were originally proposed by Hinton [11] who identified that they have recursive binding properties that allow for higher level semantic vector representations to be formulated from, and in the same format as, their lower level semantic vector components. As such they are said to be semantically self-describing. Eliasmith coined the phrase *semantic pointer*[12] for such a feature vector since it acts as both a *semantic* description of the concept, which can be manipulated directly, as well as a means of retrieving or accessing the sub-feature vectors from which it was built, i.e., it is a *'pointer'* to its 'sub-features'. Vector unbinding provides a means of retrieving the sub-feature vectors from which it was built.

VSAs are also 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)* and Logical reasoning.

They are highly resilient to noise and they have neurologically plausible analogues which may be exploited in future distributed cognitive architectures. Consequentially they have been used in natural language processing [13–15] and cognitive modeling [12,16].

Our hypothesis is that a Vector Symbolic Architecture can be used to define a rich and yet compact encoding that will enable the representation of: service descriptions; decentralized service and workflow discovery; distributed workflow execution. This will enable the ability to perform semantic matchmaking and reasoning on service descriptions and service compositions (i.e., workflows).

In this paper we extend the approach taken in [17] and show how binary VSAs can be used to achieve the following:

- **Scaling through recursive binding (chunking):** To address scalability, we extend VSAs using a hierarchical vector binding scheme that is capable of representing multiple levels of semantic abstraction (workflow and sub-workflows/branches) into a single vector. We demonstrate empirically, that this scheme can scale to tens of thousands of vectors while maintaining semantic matching, which is adequate for representing most workflows.
- **Encoding workflows:** We discuss and describe how VSA vectors in the form of role-filler pairs can be used to successfully encode both functional and QoS/QoE components of service descriptions. In addition we show how such service descriptions can be encoded via chunking to encode very large sequences of services.
- **Representing workflow primitives:** We extend the encoding scheme to support directed acyclic graph (DAG) workflows having one-to-many, many-to-many, and many-to-one connections.
- **Distributed discovery and orchestration:** We show how our VSA encoding scheme can be used for distributed discovery and orchestration of complex workflows. Workflow vectors are multicast to the network and participating services, extended with a simple cognitive layer that can interpret and exchange the vectors, compute their own compatibility and offer themselves up for participation in the workflow. We show how a delayed response mechanism, based on the degree of semantic match, can be used for selection of the best available micro-service for a particular workflow step based on both functional and QoS/QoE requirements while minimizing the bandwidth required for negotiation and selection of such.

The rest of the paper is structured as follows: Section 2 describes related work on QoS and QoE, time-critical systems and workflows; Section 3 presents an introduction to VSA mathematical operations and VSA structures; Section 4 describes and discuses how VSA can be used to build semantic representations of services and their QoS; Section 5 outlines how VSA can encode complex workflows and explains how decentralized workflow execution is performed; Section 6 describes an architecture for adding a cognitive service layer to existing services and Section 7 discusses the implementation details of the architecture; Section 8 describes the test-cases used to evaluate of our VSA workflow architecture; Section 9 describes the outcome of the test-case evaluation; Section 10 details an empirical experiment showing that the binding method supports semantic comparison of high-level workflow vectors containing many thousands of sub-feature vectors; Finally, Section 11 concludes with a short summary and a discussion of the future directions of this research.

2. Related work

2.1. Service matchmaking and optimization

With the proliferation of Web service applications, the need for methods to allow consumers to differentiate among providers of Web services was needed. The approaches adopted generally have involved identifying metrics relating to a service provider's *claimed* Quality of Service (QoS), and from a consumer's point of view, their subjective Quality of Experience (QoE) [18]. General QoS metrics such as Number of Processors, Memory size, Time-to-Execute, and Reliability (up-time) help consumers make informed decisions. For eCommerce environments, QoS metrics, such as Cost of Service, Compensation Rate, and Penalty Rate are considered paramount because they directly affect a consumer's bottom line [18,19].

QoE metrics such as Service-Ranking are more difficult to quantify and collect since they are metrics measured from the consumers' perspective and are often subjective. In transient edge environments, the lack of a queryable central repository for collecting such metrics makes these 'traditional' QoE metrics difficult to conceptualize and so we focus mainly on supporting QoS metrics in this paper. However, we discuss the idea that repeated successful participation in a distributed workflow by a component service may be a useful alternate analog of QoE in edge environments.

The majority of existing time-critical systems store such QoS and QoE attributes in centralized repositories that require a stable endpoint and typically offer matchmaking and optimization algorithms which are also managed centrally. As an example, the ATLAS experiment at CERN uses the TAG Service Catalog to collect QoS and QoE metrics from the various searches and data retrieval services sites [20]. A formal ontology is used to define the ATLAS QoS and QoE metrics and a multi-objective optimization algorithm is used with those metrics to resolve any conflicting requirements and to enable distributed service load balancing to be achieved [21]. In [19], an algorithm for normalizing multiple QoS, QoE metrics into a single value is described. The individual metrics along with the normalized QoS and QoE score are published in a central registry. Both [19] and [21] rely on a such centralized view for the QoS and QoE calculations, however time critical services operating in a MANET type environment must determine their utility based solely on local information. This requires new methods for computing similar normalized QoS and QoE metrics from local knowledge.

2.2. Time-critical systems

Time-critical systems generally rely on a global view and central management. For example, ARCADIA [22] offers centrally managed multi-infrastructure deployment. MODAClouds [23] enables the development of time-critical cloud applications but does not support software defined networking. However, SWITCH – the Software Workbench for Interactive, Time-Critical and Highly self-adaptive applications – provides a full stack solution to support the entire lifecycle of time-critical applications, and associated QoS and QoE constraints. SWITCH provides a Web programming workbench that can be used to orchestrate an application as a set of components and connections that specify QoS and QoE constraints, along with planning and runtime monitoring engines that can deploy cloud applications. It also enables runtime application reconfigurability in order to adapt to changing conditions. SWITCH is designed for cloud environments and provides a centralized coordinator for the deployment, but has a decentralized architecture for monitoring, with one monitoring component being deployed per application. SWITCH offers an interchange format (i.e., TOSCA) that is similar to our VSA model in that it provides a single format for storing application dependencies, and QoS and QoE attributes for passing around between the various parts of the system; however, it differs from VSA because it is a centralized model with reliance on a TCP-based infrastructures and it does not support semantic matching capabilities.

2.3. Workflows

Workflows provide a robust means for describing applications consisting of control and data dependencies along with the logical reasoning necessary for distributed execution. For fixed networks, there have been a wide variety of workflow systems developed [24–33]. A scientific workflow is a set of interrelated computational and data-handling tasks designed to achieve a specific goal. It is often used to automate processes which are frequently executed, or to formalize and standardize processes. On the other hand, on-demand distributed analytics workflows for general collaborative environments need spontaneous discovery of multiple distributed services without central control [6]. 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.

The authors of the decentralized workflow system – Newt – have observed similar technical challenges [34]; Newt addresses such issues for dynamic heterogeneous wireless networks. Newt is capable of processing distributed complex causal processing and interactions—in [34], the authors demonstrate this by an orchestration of William Shakespeare's play, Hamlet, constructed as a decentralized workflow, in which the individual actors are services that are distributed across a wireless network and converse by messages communicated between one actor and another as the play progresses. Our work differs because Newt does not provide discovery or semantic matchmaking capabilities, and cannot support QoS and QoE to support the coordination of time-critical cooperative workflows. In Section 8.1, we use the Hamlet play to compare and contrast our approach.

3. Vector symbolic architecture, basic operations

VSAs use hyper-dimensional vector spaces in which the vectors can be real-valued, such as in Plate's Holographic Reduced Representations (HRR) [7], typically having N dimensions ($512 \leq N < 2048$), or they can be large binary vectors, such as Pentti Kanerva's Binary Spatter Codes (BSC) [9], typically having $N \geq$ 10,000. For the work here, we have chosen to use Kanerva's BSC but we note that most of the equations and operations discussed should also be compatible with HRRs [16].

Typically, when using BSC, a basic set of symbols (e.g., an alphabet) are each assigned a fixed, randomly generated hyper-dimensional binary vector. Due to the high dimensionality of the vectors the basic symbol vectors are uncorrelated to each other with a very high probability. Hence, they are said to be *atomic* vector symbols [9]. Vector *superposition* is then used to build new vectors that represent higher level concepts (e.g., words) and these vectors in turn can be used to recursively build still higher level concepts (e.g., sentences, paragraphs, chapters...). These higher level concept vectors can be compared for similarity using a suitable distance measure such as Normalized Hamming Distance (HD).

HD is defined as the number of bit positions in which two vectors differ, divided by the dimension N of the vector. When setting bits randomly, the probability of any particular bit being set to a 1 or 0 is 0.5; hence, when generating very large random vectors, the result will be, approximately, an equal, 50/50, split of

1s and 0s distributed in a random pattern across the vector. Thus, when comparing any two such randomly generated vectors, the expected HD will be $HD \approx 0.5$. Indeed, for 10 kb binary vectors, the probability of two randomly generated vectors having a HD closer than 0.47 (i.e., differing in only 4700 bit positions instead of approximately 5000) is less than 1 in 10^9 [9, page 143]. For the same reason; *atomic* vectors can be generated as needed, on the fly, without fear that the newly generated random vector will be mathematically similar to any existing vector in the vector space. Further, by implication, when using HD on BSC to test for similarity, a threshold of 0.47 or lower implies a match has been detected with a probability of error $\leq 10^{-9}$. A threshold of 0.476 or lower implies a match with a probability of error of $\leq 10^{-6}$. Thus, in our experiments, we used 0.47 as the threshold.

For BSCs, *superposition* is archived using *bitwise majority voting*, a form of vector addition [9]. Simply put, for any particular column of bits in the sum, the majority wins; ties are broken randomly. Mathematically, when summing n vectors V, for any bit position i, set the corresponding output bit X_i as follows,

$$X[i] = \begin{cases} 1, & \text{if } \left(\sum_{j=1}^{n} V_j[i]\right)/n > 0.5 \\ 0, & \text{if } \left(\sum_{j=1}^{n} V_j[i]\right)/n < 0.5 \\ random, & \text{if } \left(\sum_{j=1}^{n} V_j[i]\right)/n = 0.5 \end{cases} \quad (1)$$

The resulting vector is of equal size to its sub-feature vectors and represents the lossy *superposition* of these components such that each vector element in the result participates in the representation of many entities, and each entity is represented collectively by many elements of the resultant vector [16].

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

$$\mathbf{Person1_v} = \mathbf{John_v} + \mathbf{Charles_v} + \mathbf{55yrs_v} + \mathbf{T2Diabetic_v}$$
$$\mathbf{Person2_v} = \mathbf{Lucy_v} + \mathbf{Charles_v} + \mathbf{55yrs_v} + \mathbf{T2Diabetic_v}$$
$$\mathbf{Person3_v} = \mathbf{Greg_v} + \mathbf{Charles_v} + \mathbf{34yrs_v} + \mathbf{T2Diabetic_v}$$

where + is defined as the *bitwise majority vote* operator. HD can be used to compare such vectors *without* unpacking or decoding the sub-features. Using HD to compare $Person1_v$[1] with $Person2_v$ will give a match since they have 3 common sub-features. Also, $Person1_v$ and $Person2_v$ are more similar to each other than they are to $Person3_v$. An issue arises, however, when using superposition to build compound vectors in this way because such compound vectors behave as an unordered *bag of features*. Thus, if we have,

$$\mathbf{Person4_v} = \mathbf{Charles_v} + \mathbf{Smith_v} + \mathbf{55yrs_v} + \mathbf{T2Diabetic_v}$$

Then, $Person4_v$ would be equally similar to $Person1_v$ as is $Person2_v$ despite the obvious difference in the record.

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

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

For example, the variable *deposit_amount* might play the role of dollars being deposited in a banking transaction program.

When an atomic *role* vector is bound to a vector value this results in a *role-filler* pair which is analogous to variable assignment in conventional programming. For example, the statement *deposit_amount* = 300 is said to *bind* the value 300 to the variable *deposit_amount*. In a similar way, feature values such as $Charles_v$ can be bound to a role vector and detected or extracted from the *role-filler* pair vector using an inverse binding operator. Bitwise XOR is used for both *binding* and *unbinding* with BSC because it is its own inverse—i.e., BSC is commutative and distributive over superposition as well as being invertible [9, page 147]. This means that both *roles* and *fillers* can be retrieved from a *role-filler* pair without any loss. For example, if $\boldsymbol{Z} = \boldsymbol{X} \cdot \boldsymbol{A}$ then $\boldsymbol{X} \cdot \boldsymbol{Z} = \boldsymbol{X} \cdot (\boldsymbol{X} \cdot \boldsymbol{A}) = \boldsymbol{X} \cdot \boldsymbol{X} \cdot \boldsymbol{A} = \boldsymbol{A}$ since $\boldsymbol{X} \cdot \boldsymbol{X} = \mathbf{0}$ (i.e., the zero vector) where '$\cdot$' represents the bitwise XOR operator. Similarly, $\boldsymbol{A} \cdot \boldsymbol{Z} = \boldsymbol{X}$.

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

$$\boldsymbol{Z} = \boldsymbol{X} \cdot \boldsymbol{A} + \boldsymbol{Y} \cdot \boldsymbol{B} \quad (2)$$

$$\boldsymbol{X} \cdot \boldsymbol{Z} = \boldsymbol{X} \cdot (\boldsymbol{X} \cdot \boldsymbol{A} + \boldsymbol{Y} \cdot \boldsymbol{B}) = \boldsymbol{X} \cdot \boldsymbol{X} \cdot \boldsymbol{A} + \boldsymbol{X} \cdot \boldsymbol{Y} \cdot \boldsymbol{B} \quad (3)$$

$$\boldsymbol{X} \cdot \boldsymbol{Z} = \boldsymbol{A} + \boldsymbol{X} \cdot \boldsymbol{Y} \cdot \boldsymbol{B} \quad (4)$$

Examination of Eq. (4) reveals that vector **A** has been exposed, thus, if we perform $\boldsymbol{HD}(\boldsymbol{X} \cdot \boldsymbol{Z}, \boldsymbol{A})$ we will get a match. The second term $\boldsymbol{X} \cdot \boldsymbol{Y} \cdot \boldsymbol{B}$ is considered noise because $\boldsymbol{X} \cdot \boldsymbol{Y} \cdot \boldsymbol{B}$ is not in our known *vocabulary* of features or symbols.

When a role and value are bound together this is equivalent to performing a mapping or *permutation* of a vector's value elements within the hyper-dimensional space so that the new vector produced is uncorrelated to both the role and filler vectors. For example, if $\boldsymbol{V} = \boldsymbol{R} \cdot \boldsymbol{A}$ and $\boldsymbol{W} = \boldsymbol{R} \cdot \boldsymbol{B}$ then $\boldsymbol{R}$, $\boldsymbol{A}$ and $\boldsymbol{B}$ will have no similarity to $\boldsymbol{V}$ or $\boldsymbol{W}$. However, comparing $\boldsymbol{V}$ with $\boldsymbol{W}$ will produce the same match value as comparing $\boldsymbol{A}$ with $\boldsymbol{B}$. In other words, if $\boldsymbol{A}$ is closely similar to $\boldsymbol{B}$ then $\boldsymbol{V}$ will be closely similar to $\boldsymbol{W}$ because *binding* preserves distance within the hyper-dimensional space [9, page 147].

We note that *binding* with *atomic* role vectors can be used as a method of *hiding* and *separating* values within a compound vector whilst maintaining the comparability between compound vectors. This is an important property and can be used to encode position and temporal information about sub-feature vectors within a compound vector. It also explains why we can state that $\boldsymbol{X} \cdot \boldsymbol{Y} \cdot \boldsymbol{B}$ from Eq. (4) above will not match to any known symbol, however, note that we can get back to $\boldsymbol{B}$ from $\boldsymbol{X} \cdot \boldsymbol{Y} \cdot \boldsymbol{B}$ by simply performing the appropriate XOR—i.e., $\boldsymbol{B} = ((\boldsymbol{X} \cdot \boldsymbol{Y} \cdot \boldsymbol{B}) \cdot \boldsymbol{X}) \cdot \boldsymbol{Y}$.

We can now rephrase our *person* record in order to differentiate sub-features within the record, for example, we can formulate $\boldsymbol{Person1_v}$ as:

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

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

$$\boldsymbol{HD}(\boldsymbol{SN_r} \cdot \ \boldsymbol{Person1_v}, \ \boldsymbol{Charles_v}) \quad (5)$$

For 10 kb vectors, if the result of Eq. (5) is less than 0.47 then the probability of $\boldsymbol{Charles_v}$ being detected in error is less than 1 in 10^9 [9, page 143]. If our *person* record is distributed over a network we could transmit or multicast the request vector $\boldsymbol{Z} = \boldsymbol{SN_r} \cdot \boldsymbol{Charles_v} + \boldsymbol{Age_r} \cdot \boldsymbol{55years_v}$ to the network. Any listening distributed micro-service, or node in a Parallel Distributed Processing network, having person records containing the surname

$\boldsymbol{Charles_v}$ and age $\boldsymbol{55years_v}$ can check for a match and respond or become activated.

Since binding and superposition are such simple operations, we note that a key advantage of this approach is that complex representations of services can be built using very simple knowledge engineering approaches as described in Section 4. Further, simultaneous comparisons of complex objects are reduced to a single Hamming Distance calculation which greatly simplifies service discovery/match making as compared to traditional ontology based approaches.

4. Building semantic vector representations of services and QoS

Having shown that we can use VSAs to represent data structures, we now consider how to represent service descriptions and their corresponding QoS as symbolic vectors. Reviewing that $\boldsymbol{X_r}$ represents an *atomic* role vector and $\boldsymbol{Y_v}$ a value vector and that $\boldsymbol{X_r} \cdot \boldsymbol{Y_v}$ is a role-filler pair that binds the category $\boldsymbol{X_r}$ to the filler value $\boldsymbol{Y_v}$ and enables later matching and retrieval of values by specific categories, our current scheme employs the following format:

$$\boldsymbol{Z_x = Serv_r \cdot Serv_v + Resource_r \cdot ResP_v + QoS_r \cdot QoS_v} \tag{6}$$

where

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

4.0.1. Building the description vector

$\boldsymbol{Serv_v}$ is itself comprised of symbolic vectors that semantically describe the essential elements of a service, in terms of role and filler pairs that are needed to find a match. To illustrate how this is achieved we use an example of relatively simple service description comprising service name, inputs, outputs, and a functional description of the service, for example:

$$\boldsymbol{Serv_v = Inputs_r \cdot Inp_v + Name_r \cdot Name_v + Desc_r \cdot Desc_v + Outputs_r \cdot Out_v} \tag{7}$$

Where

- $\boldsymbol{Inputs_r \cdot Inp_v}$ describes the required inputs;
- $\boldsymbol{Name_r \cdot Name_v}$ a vector encoding of the service name;
- $\boldsymbol{Desc_r \cdot Desc_v}$ a vector encoding of the service description[2]; and
- $\boldsymbol{Outputs_r \cdot Out_v}$ describes the required outputs.

Again the filler component of these vectors can be comprised of other symbolic vectors. In considering $\boldsymbol{Inp_v}$, $\boldsymbol{Out_v}$ we want to encode these values so that we get flexible matching. For example, if our service, $\boldsymbol{Z_x}$, has three float inputs and one bitmap input we might encode this as:

$$\boldsymbol{Inp_v = One_r \cdot Float_r + Two_r \cdot Float_r + Three_r \cdot Float_r + One_r \cdot BitMap_r} \tag{8}$$

[2] Currently, we can build vector representations using JSON or XML description of services.

$\boldsymbol{One_r}$, $\boldsymbol{Two_r}$, $\boldsymbol{Three_r}$ are atomic role vectors representing numbers. This simple scheme seems adequate for representing input and output descriptions because micro-services typically do not have a large number of inputs and outputs. More complex input and output descriptions can be encoded via embedding further role-filler pairs. The above vector is a bag representing the inputs that enables flexible matching. If the input part of a request vector is encoded as:

$$\boldsymbol{InpReq_v = One_r \cdot Float_r + One_r v \cdot BitMap_r}$$

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

4.0.2. Building the quality of service vector

For QoS, we employ a slightly different encoding scheme. For example, a QoS metric often has a requirement to meet a certain a minimum or maximum value for the metric in question. An example of a static QoS metric might be that the service must possess a minimum of four CPU cores. A simple way to encode minimum or maximums, so that they are semantically comparable, is in the form of a bag of acceptable values. In the number of cores example, to specify four or more cores we can encode:

$$\boldsymbol{CpuCores_r \cdot (Four_r + Five_r + Six_r + \cdots + MaxCores_r)}$$

Therefore, an individual service that encodes its *CpuCores* QoS as $\boldsymbol{CpuCores_r \cdot Four_r}$ or $\boldsymbol{CpuCores_r \cdot Eight_r}$ would be a match. For time-critical distributed applications, *available compute power* might be a better QoS and could be a normalized value combining number of CPUs and GPUs, memory, clock-speed along with the current load. In order to facilitate semantic comparisons of such a metric, a service calculating its value would then quantize it to the nearest higher or lower value depending on if the expected comparison is a max or min requirement. Dynamic QoS metrics such as *battery life percentage* or *available runtime* can also be encoded in this way. For example, aggregate bandwidth, obtained by each service actively monitoring its local bandwidth with pings, might be quantized to **(1 Kb, 10 Kb, 100 Kb, 1 Mb, 10 Mb, 100 Mb, 1 Gb)/s**. Such values are then converted to an enumeration thereby allowing us to encode ranges that represent different underlying values with the same role vectors. Thus, a minimum bandwidth QoS requirement of, say, 100 Mb/s (100 Mb is in 6th position in the above list) would be encoded as follows:

$$\boldsymbol{Bandwidth_r \cdot (Six_r + Seven_r)}$$

The above describes our current method for encoding service descriptions and QoS as BSC vectors. VSA superposition allows us to combine any set of individual functional and QoS parameters into a bag of features for simultaneous comparison and matching enabling a far more flexible approach than the ontology-style approach. Both [19] and [21] describe methods that can be used to combine multiple QoS metrics into a single normalized value that reflects the weighting given to each individual metric. We are currently investigating how best to encode this type of metric using our VSA representation.

5. Describing workflows using vector symbolic architecture

As discussed in Section 3, VSAs employ two operations—i.e., *binding* and *superposition*. *Binding* is used to build *role-filler* pairs which allow sub-feature vectors to remain separate and identifiable (although hidden) when bundled into a compound

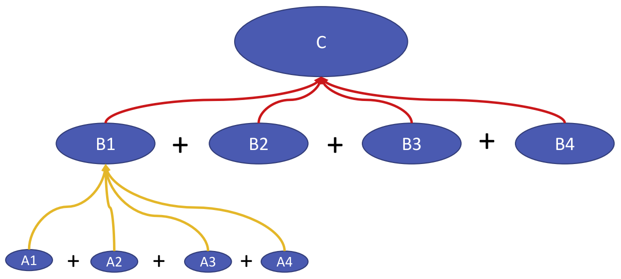

Fig. 1. Vector chunk tree, chunking proceeds from the bottom up.

vector via *superposition*. For BSCs, *binding* is a lossless operation, while *superposition* is lossy. Kleyko [10, Paper B, page 80] supplies a mathematical analysis of the capacity of a single compound vector such that it can be reliably unbound, i.e., its sub-feature vectors can be reliably detected within the compound vector. This analysis shows that for 10 kb binary vectors the upper limit of superposition is 89 sub-vectors. To encode large workflows with more complex service descriptions we require a method for combining more vectors into a single vector whilst maintaining the semantic matching properties.

Chunking is a recursive binding method that combines groups of vectors into a single compound vector. The resultant vectors are then used as the basis for further chunking operations, thus, recursively producing a hierarchical tree structure as shown in Fig. 1. Chunking proceeds from the bottom up so that each node in the tree is a compound vector encapsulating the child nodes from the level below. Various methods of recursive *chunking* have been described [7,9,10,16]. However, such methods suffer from limitations when employed for multilevel recursion: some lose their semantic matching ability even if only a single term differs, others cannot maintain separation of sub-features for higher level compound vectors when lower level chunks contain the same vectors [9, page 148] [7, pages 61, 72, 74–para2] [10, Encoding Sequences, page 14]. We addressed these issues and describe a novel recursive encoding scheme that provides semantic matching at each level by combining two different methods of permuting vectors.

In our scheme, the terminal nodes are worker services, the higher level nodes are concepts used to apply grouping to parts of the workflow. The chunking process occurs from the bottom up so that the bottom level nodes, $\{\boldsymbol{A1}, \boldsymbol{A2}, \boldsymbol{A3}, \ldots\}$ are combined via a functional partitioning scheme into a concept node, e.g., $\boldsymbol{B1}$. These higher level nodes – referred to as *clean-up memory* [7,9,10] – are still services but they simply provide a proxy to the worker services to be unbounded and executed; thus, they are typically co-located with the first service of the sub-sequence they represent, e.g., $\boldsymbol{B1}$ can reside on the same compute node as the $\boldsymbol{A1}$ service and hence, when $\boldsymbol{B1}$ becomes activated, there is no need for a network transmission in order for $\boldsymbol{B1}$ to activate its first worker service, $\boldsymbol{A1}$. In a centralized system, *Clean-up memory* is typically implemented as an auto-associative memory. For our distributed workflow system, clean-up memory is implemented by the services themselves, which are distributed throughout the network, matching and resolving to their own vector representations.

Recchia and Kanerva point out that for large random vectors, any mapping that permutes the elements can be used as a binding operator, including cyclic-shift [15]. The encoding scheme shown in Eq. (9) employs both XOR and cyclic-shift binding to enable recursive bindings capable of encoding many thousands of sub-feature vectors even when there are repetitions and similarities between sub-features:

$$\boldsymbol{Z_x} = \sum_{i=1}^{cx} \boldsymbol{Z_i^i} \cdot \prod_{j=0}^{i-1} \boldsymbol{p_j^0} + \boldsymbol{StopVec} \cdot \prod_{j=0}^{i} \boldsymbol{p_j^0} \tag{9}$$

Omitting $\boldsymbol{StopVec}$ for readability, this expands to,

$$\boldsymbol{Z_x} = \boldsymbol{p_0^0} \cdot \boldsymbol{Z_1^1} + \boldsymbol{p_0^0} \cdot \boldsymbol{p_1^0} \cdot \boldsymbol{Z_2^2} + \boldsymbol{p_0^0} \cdot \boldsymbol{p_1^0} \cdot \boldsymbol{p_2^0} \cdot \boldsymbol{Z_3^3} + \cdots \tag{10}$$

Where

- $\cdot$ is defined as the XOR operator;
- $+$ is defined as the Bitwise_Majority_Vote/Add operator;
- The exponentiation operator is redefined to mean cyclic-shift—i.e., positive exponents mean $\boldsymbol{C_{shift\_right}}$, negative exponents mean $\boldsymbol{C_{shift\_left}}$. Note that cyclic shift is key to the recursive binding scheme since it distributes over $+$ (i.e., bitwise majority addition) and $\cdot$ (i.e., XOR) hence it automatically promotes its contents into a new part of the hyper-dimensional space; thus, keeping levels in the chunk hierarchy separate;
- $\boldsymbol{Z_x}$ is the next highest semantic *chunk* item containing a *superposition* of $\boldsymbol{x}$ sub-feature vectors. $\boldsymbol{Z_x}$ chunks can be combined using Eq. (9) into higher level chunks. For example, $\boldsymbol{Z_x}$ might be the superposition of $\boldsymbol{B1} = \{\boldsymbol{A1}, \boldsymbol{A2}, \boldsymbol{A3}, \ldots\}$ or $\boldsymbol{C} = \{\boldsymbol{B1}, \boldsymbol{B2}, \boldsymbol{B3}, \ldots\}$;
- $\{\boldsymbol{Z_1}, \boldsymbol{Z_2}, \boldsymbol{Z_3}, \ldots, \boldsymbol{Z_n}\}$ are the sub-feature vectors being combined for the individual nodes of Fig. 1. Each $\boldsymbol{Z_n}$ itself can be a compound vector representing a sub-workflow or a complex vector description for an individual service step, built using the methods described in Section 4;
- $\boldsymbol{p_0}, \boldsymbol{p_1}, \boldsymbol{p_2}, \ldots$ are a set of known *atomic* role vectors used to define the current position or step in the workflow.
- $\boldsymbol{cx}$ is the chunk size of vector $\boldsymbol{Z_x}$, i.e., the number of sub-feature vectors being combined; and
- $\boldsymbol{StopVec}$ is a role vector owned by each $\boldsymbol{Z_x}$ that enables it to detected when all of the steps in its (sub)workflow have been executed.

5.1. Ordered unbinding of high-level concept vectors

Eq. (9) is used recursively to build a workflow request, conceptually creating a hierarchical chunk tree as shown in Fig. 1. The resulting output is a set of VSA vectors representing the non-terminal nodes, $\{\boldsymbol{C}, \boldsymbol{B1}, \boldsymbol{B2}, \boldsymbol{B3}, \ldots\}$, each of which is a single VSA vector, $\boldsymbol{Z_x}$ that is itself a compound vector representing the ordered sequence of its own children.

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

The reason multiple $\boldsymbol{p}$ vectors are XOR chained together to define a single position within the workflow is due to the distributive property of XOR which operates on every term for each unbinding. Thus, the use of the chained $\boldsymbol{p}$ vectors when constructing the workflow vector allows for easier iterative unbinding at execution time using Eq. (13), discussed below. Note that the distributive affect of XOR can also be seen in how the $\boldsymbol{T}$ vector becomes permuted during unbinding, see Eq. (12) and Eq. (14).

The generalized version of the concept vectors, $\{\boldsymbol{C}, \boldsymbol{B1}, \boldsymbol{B2}, \ldots\}$, is shown in Eq. (10). Note that, in this form every substep $\boldsymbol{Z_n}$ is permuted by at least one $\boldsymbol{p}$ vector which effectively hides each $\boldsymbol{Z_n}$ (the $\boldsymbol{p}$ vector permutation ensures that each combined roll-filler pair is orthogonal to the 'self-description' vectors built by each VSA service listening for work on the network). The workflow is discovered and orchestrated on the distributed services by, essentially, repeatedly *unbinding* the workflow vector, using Eq. (11) or Eq. (13), before retransmitting it to the network.

Referring to Fig. 1, control first passes down the chunk tree, i.e., from $\boldsymbol{C} \rightarrow \boldsymbol{B1} \rightarrow \boldsymbol{A1}$ using Eq. (11), before traversing horizontally, $\boldsymbol{A1} \rightarrow \boldsymbol{A2} \rightarrow \boldsymbol{A3} \rightarrow \boldsymbol{A4} \rightarrow \boldsymbol{B1\_StopVec}$) via Eq. (13). At this point $\boldsymbol{B1}$ *sees* its $\boldsymbol{StopVec}$ and employs Eq. (13) to activate $\boldsymbol{B2}$ which then activates its sub-workflow via Eq. (11) and so forth.

Starting a (sub)workflow :

$$\boldsymbol{Z'_1 = (p_0^0.(T + Z_x))^{-1}} \tag{11}$$

$$\boldsymbol{Z'_1 = p_0^{-1}.T^{-1} + \boxed{Z_1^0} + p_1^{-1}.Z_2^1 + p_1^{-1}.p_2^{-1}.Z_3^2 + \cdots} \tag{12}$$

Traversing horizontally :

$$\boldsymbol{Z'_{n+1} = (p_n^{-n} \, . \, Z'_n)^{-1}} \tag{13}$$

$$\boldsymbol{Z'_2 = (p_1^{-1}.Z'_1)^{-1} = p_1^{-1}.p_0^{-2}.T^{-2} + p_1^{-1}.Z_1^{-1} + \boxed{Z_2^0} + p_2^{-2}.Z_3^1 +} \tag{14}$$

When starting a (sub)workflow using Eq. (11) notice that $\boldsymbol{Z_1}$ has been exposed, as shown in Eq. (12). That is, $\boldsymbol{Z'_1}$ is effectively a noisy copy of the currently required workflow step, $\boldsymbol{Z_1}$, while at the same time it is also a 'masked' description of the full (sub)workflow request. Thus, listening services can only match to $\boldsymbol{Z'_1}$ if they are semantically similar to $\boldsymbol{Z_1}$. Note that, the act of matching gives a service no other information; for example, it cannot deduce by matching alone whether the match occurred at step 1 or step 30 of the workflow. Hence, the introduction of the $\boldsymbol{T}$ vector in Eq. (11) which is used to enable calculation of a node's position within the workflow.

The $\boldsymbol{T}$ vector is a known *atomic* role vector. It is added to a high level node's, clean, (sub)workflow vector, $\boldsymbol{Z_x}$, before the node uses Eq. (13) to expose its first workflow step for transmission to the network. We note that, Eq. (11) is just a special version of Eq. (13). Notice in Eqs. (12) and (14), how the $\boldsymbol{T}$ vector becomes permuted in a predictable way. Once the currently active service has completed its own workflow step it uses the current permutation of the $\boldsymbol{T}$ vector to calculate its position $\boldsymbol{n}$ within the received request vector. It can then activate the next workflow step in the request by repeating the *unbind* operation on the request vector, generalized in Eq. (13). Thus, the workflow proceeds in a completely decentralized manner whereby each node is activated when its preceding node, or parent, *unbinds* the currently active chunk vector, creating the next request vector, which it then multicasts to the network for matching and processing.

Alternative mechanisms for determining the position of the service are possible but these require each service to recursively unbind all vectors that it receives to determine if it is part of the requested workflow. This significantly increases the work that each service has to perform which is undesirable. However, such a mechanisms may offer some advantages as discussed in the following section.

5.2. Pre-provisoning and learning to get ready

From Eq. (13) we see that each workflow step is exposed by iterative application of p vector permutations. Non-matching services can use this method to *peek* a vector enabling anticipatory behavior such as the pre-provisioning of a large data-set or changing a device's physical position (e.g., drones). Obviously, services can *peek* multiple steps into the future and could *learn* how early to start pre-provisioning. This ability to anticipate could be used to perform more complex, on-line, utility optimization learning. For example, a drone monitoring multiple workflows may be able to understand that it will be needed in 10 min to perform a low priority task and in 15 min for a high priority task. Under these circumstances it may choose not to accept the low priority task.

5.3. Alternate to QoE for distributed transient environments

As can be seen in Eqs. (12) and (14), when a particular workflow step is exposed for discovery and execution by unbinding, it is 'surrounded by' (i.e., it is in superposition with) the rest of the workflow steps which, as can be seen, are permuted in a specific way depending on the position of currently exposed/active service in the workflow. We can think of this as the current permutation of the workflow vector and it constitutes a context for the workflow step currently in focus. We are investigating the use of these contexts as an analog of QoE. The idea is that when a services successfully participates in a workflow it will remember the permutation state of the workflow vector via which it was activated. If a particular service successfully participates in the same workflow repeatedly; these workflow context memories can be used to increase the particular service's utility with respect to the specific workflow. We suggest that this might be an interesting analog of traditional QoE measures which we believe would be almost impossible to measure and collect in distributed transient environments. The idea is that a service that often helps complete a particular workflow should be seen as a more valuable partner by the other service steps, hence an analog of QoE.

6. Decentralized architecture for time-critical applications

This section describes an architecture that enables linear or Directed Acyclic Graph (DAG) workflows, comprising multiple interconnected services, to be configured with no central point of control. As discussed in Sections 4 and 5, a workflow can be constructed as a composite symbolic vector that is itself built from the symbolic vectors that describe the component services in terms of capability and utility, and from symbolic vectors that describe the links between the component services. In Section 8 we describe how more complex workflows are represented. The decentralized architecture requires mechanisms for the construction and transmission of these vectors and for services to be 'cognitively enabled, so that they can participate in any required workflow. Our architectural approach to this challenge is achieved by adding to existing component services a cognitive layer, using the symbolic vector representation, that enables services to be self-describing and to self-organize into the requested distributed workflow.

At a high level, the architecture supports the automatic generation of an application vector (i.e., services and workflow) from an existing service/workflow description (e.g., JSON or XML). To construct the distributed application, an initiator service unbinds and transmits the application vector from anywhere in the network (e.g., using broadcast or multicast). The cognitive layer for each of the distributed service generates and maintains its own description vector based on the local service description and its current QoS (i.e., services are dynamically self-describing). The services then listen for the transmitted application vectors and reactively respond by performing logical vector operations on the received vectors. If the service vector semantically matches a received vector, it is a potential match but there may be other similar services that also match. To ensure that the best service is

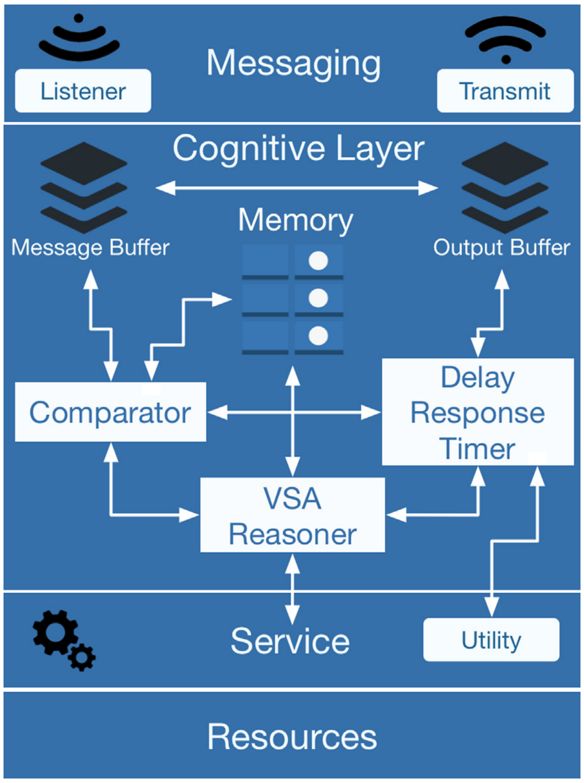

Fig. 2. Overview of the components for time-critical framework.

selected and to minimize the number of services that respond, the architecture makes use of a delayed response timer mechanism. This ensures that services with the highest utility respond first, suppressing the responses of alternative services with a lower utility. To avoid problems such as race conditions where more than one candidate service responds a local arbitration mechanism is used to select the service with highest utility. If a service is selected, it may perform some local function and when completed it unbinds the received vector and re-transmits. This will result in other services reactively responding to the new unbound vector. In this way control is passed around the network of distributed services.

To achieve these vector comparison and exchange objectives, the cognitive service layer comprises the following components:

1. **Message Listener and buffer** for received symbolic vector messages;
2. **Symbolic Vector Memory** to store vectors required for various logical operations;
3. **Comparator** to compute semantic similarity between symbolic vectors;
4. **VSA Reasoner** to perform logical operations on the symbolic vectors;
5. **Delay Response Timer** to control if and when new symbolic vectors are to be transmitted; and
6. **Message Transmitter** to transmit new symbolic vectors.

These components are described in the next subsections and the relationship of the components are given in Fig. 2.

6.1. *Message Listener and buffer*

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

6.2. *Symbolic Vector Memory*

The Symbolic Vector Memory is used to store vectors that are to be used for any operation required by the cognitive layer of the specific service. This would always include the service description vector and would typically include specific vectors used to support vector binding and unbinding operations or to support 'clean-up memory' that we describe in Section 5.1. In other examples the memory is used to store application vectors which when received on previous occasions resulted in the service being selected. These vectors essentially represent the context in which the service was historically invoked and these can be used to increase the utility (i.e., QoE) of the service if the same workflow is requested at a later time.

6.3. *Comparator*

To semantically compare vectors, we use a Hamming distance measure and declare a match if the Hamming distance is within particular ranges. The comparator uses the computed Hamming distance to determine an appropriate time delay based on the degree of the semantic match. The semantic match time delay, $\boldsymbol{t_{sm}}$, is a quantized value in steps of $\boldsymbol{\Delta t}$ (usually 10 levels) where a perfect match produces no delay and a marginal match near the Hamming threshold produces the maximum delay say $10 \times \boldsymbol{\Delta t}$.

6.4. *VSA reasoner*

The VSA Reasoner performs various operations on received symbolic vectors that exceed the Hamming threshold. These operations depend on the type of vector that is received. For example, in the case of receiving an unbound workflow vector that matches the service name vector, the VSA reasoner may simply unbind the received vector and pass it to the message transmit buffer. In other cases, the response to the match may be to transmit a clean version of the noisy vector that was received (clean-up memory). Additionally, the reasoner can be tasked to 'peek' a received workflow vector and to determine if and when the current service may be called in order to pre-provision the service. This task can also include listening to the progress of a particular workflow as flow control is passed among the component services to ensure that the current service has reached its maximum utility if and when it is invoked. The VSA Reasoner also includes an important sub component called the Vector Encoder which is used to compile symbolic vectors that semantically describe the supported service and its current utility. These vectors can themselves be constructed from other symbolic vectors using the hierarchical chunking scheme described in Eq. (9).

6.5. *Delay response timer*

There may be multiple services that could respond to the same workflow request. The purpose of the Delay Response Timer is to ensure that only services with the best symbolic match and hence the highest utility to perform the task will multicast a response message thereby saving bandwidth, as described in 7.2. The use of a time delay to select resources with the highest utility has previously been used successfully to control the connectivity and growth of a dynamic distributed database architecture known as the Gaian Database [35,36]. How this mechanism operates can be understood from a simple example of a service that is attempting to offer itself as a candidate to be included in a requested workflow. To do this, it needs to determine that it semantically matches the requested service and then be the first matching service to react by transmitting a *Response* vector. On reception of the workflow vector the service uses the Comparator to determine its degree of match and the corresponding time delay $\boldsymbol{t_{sm}}$. The service must also have a particular utility to perform the task which may be based on a number of factors such as available power, the compute platform that it is operating on, connectivity to other resources required for the task, and so on. The service uses its utility to compute a second time delay $\boldsymbol{t_{ut}}$ which ranges from zero where there is the highest utility rising to $\boldsymbol{\Delta}$t where the utility is low but still adequate to compute the task. If the utility is not sufficient then tut is essentially infinity and the Delay Response Timer will not allow response vector to be multicast. The Delay Response Timer now computes a total delay of $\boldsymbol{t_d = t_{sm} + t_{ut}}$ and stores this with the message in the Message Transmit Buffer.

6.6. Message transmit buffer

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

7. Implementation example

The architecture discussed in the previous section has been implemented in our VSA platform in `Python2`. The VSA platform has a modular architecture with several components that are capable of being reused as plugins to other systems. The platform is used to evaluate and demonstrate how symbolic vectors can be automatically constructed from typical scientific workflow representations and how these vectors can then be used to construct, in a decentralized manner, the required workflow in an emulated wireless network environment into which the cognitively enabled services are randomly deployed.

- **The Workflow Importer** component imports a Pegasus workflow description (DAX) file [37]. This is an, XML format, multi-nested dictionary description of a workflow which details each service node and its input output resources. The Workflow Importer reads the DAX file into a python dictionary. It then parses the dictionary and extracts the *job* entries to create a list of vectors that represent each service node in the DAX, the *NodeVectors* list. Similarly, it traverses the *child* section of the DAX producing the *EdgeVectors* list, a paired list of vectors representing the parent (output) and child (input) connections of the workflow. The Workflow Importer passes *NodeVectors* and *EdgeVectors* to the VSA Creator.
- **The VSA Creator** is used to bind the lists of vectors into a single vector, a reduced representation, of the workflow using chunking. Chunking is performed bottom up so that higher level vectors are produced as needed. These are recursively rebound until the vector list is reduced to a single vector value. The *NodeVectors* list and the *EdgeVectors* list are combined separately producing two high level vectors, the $\boldsymbol{Recruit_{Nodes}}$ vector and the $\boldsymbol{Connect_{Nodes}}$ 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 as per Fig. 1 which is then passed to the VSA executor.
- **The VSA Executor** *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.
- **The Cognitive Layer** consists of the following sub-components
  - **The VSA Reasoner** is responsible for unbinding VSA vectors and depending on the sub-features of what it has received, it acts accordingly.
  - **The Comparator** is a Hamming distance function that performs the comparison of vectors to perform the matching.
  - **Delay Response Time Engine** is responsible for calculating the fitness function and initiates a delay timer, based on the resulting utility.
- **Messaging System** which provides the communications interface for transmitting and listening for vector communications to and from other nodes, along with internal buffers for synchronization with the other system components.
- **The Logging Component** collects metrics as the workflow runs to feed into external processors. Logging currently collects a trace of the nodes and edges that are being processed by the workflow.
- **The Visualization Component** takes the log output and generates a DAG layout graph using Graphviz [38].

7.1. Control operations

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

The current implementation of the **Cognitive Service Layer** can operate in one of two modes. In the first or dynamic mode, the workflow is required to instantiate and run the associated services as the workflow unbinding progresses. This mode is applicable to linear workflows. In the second or static mode the vector unbinding is used to gather and connect the services into a workflow configuration that is then initiated to perform the required task. This is applicable to more complex workflows with branching and merging requirements. In Sections 8.1 and 8.2

examples of both linear and complex workflow use-cases are given.

7.2. Service selection by local arbitration

A major advantage of the VSA approach is the ability to discover and select services using semantic matching and we have shown that this can extend beyond simple matches to include measures of real time utility. Service selection therefore involves selecting the correct service with the highest utility or, if the service is not available, suggesting the nearest semantically matching service. For time critical applications that need to be resilient to changes in network connectivity robustness can be achieved by distributing multiple copies of services throughout the communications network. Further, within the constraints of our target environment – i.e., field operations in very transient, low bandwidth MANETs – it is critical that we do not use unnecessary bandwidth in order to obtain an optimal solution if a sub-optimal solution meets the requirement. Using our delay-response mechanism largely addresses this by reducing the number of services that respond, however network latency sometimes results in situations where multiple services, with a similar semantic match, respond. A method of preventing race conditions in these circumstances is required. We have termed this process *local arbitration*. It is described in Section 7.4, steps 4–8, Note that, while the currently active service behaves as the *final arbiter* for services that do send a match response, the process of local arbitration is actually a distributed process. Matching services can and do exclude themselves, as described in Section 7.4, steps 6 and 7, without ever transmitting their match response.

In our implementation this is achieved as follows. Using terminology from Eqs. (11) and (13), if the currently active service is $\boldsymbol{Z'_n}$, then before transmitting the next service request, it enters *match collecting mode* in order to arbitrate matches from all nodes that reply within a tunable window of time. After the interval expires the highest ranking responder is selected and a *continue* message is broadcast by $\boldsymbol{Z'_n}$ identifying the winner. Since all communications is multicast, all services see all messages, and consequently the winning service continues and losing services discontinue. To reduce communication overhead further matching services delay their response by an interval inversely proportional to their match value as described in Section 6.5. Thus, better matches respond quicker. If a service *sees* a higher match value before it has responded then it terminates without sending a reply. Response and Continue vectors are encoded using the currently active workflow vector as a *tag-id* as follows

$$\mathbf{Response_v = Response_r \cdot Z'_n + MyID_r \cdot ID_v + Match_r \cdot Match_v} \tag{15}$$

$$\boldsymbol{Continue_v = Response^1_v} \tag{16}$$

7.3. Pre-provisioning

Although the current architecture does not support the type of pre-provisioning described in Section 5.2 it would be possible to add an extra step into the control operations between step 1 and 2 where on the receipt of any vector the layer unbinds multiple times to determine if the local service name is part of the requested workflow. If it is the case, then it can determine where in the workflow it is going to be called and start to pre-provision. When at some later time it receives the matching unbound vector its utility will be higher and it will respond faster, making it more likely that it will be the winning service.

7.4. Dynamic workflow control

In this subsection, we provide a step-by-step account of how a workflow is instantiated as a result of the unbinding progresses. The steps are as follows:

1. Compile local service vector $\boldsymbol{Z_s}$ and utility vector $\boldsymbol{Z_u}$
2. Listen and receive $\boldsymbol{Z'_N}$
3. Compare service component of $\boldsymbol{Z'_N}$ (i.e., $\boldsymbol{Z'_N \cdot Serv_r}$) with local service vector to compute the semantic match and if there is a match compute time delay *tsm* based on the Hamming distance. If no match return to listening for new vectors.
4. Compare utility component of $\boldsymbol{Z'_N}$ (i.e., $\boldsymbol{Z'_N \cdot Util_r}$) with local utility vector and if a match compute time delay *tut* based on the Hamming distance. If no match return to listening for new vectors.
5. Compute response vector $\boldsymbol{R_N}$
6. Listen for $\boldsymbol{R_N}$ equivalent vectors from other fitter services for period $\boldsymbol{td = tsm + tut}$. If non heard, then transmit $\boldsymbol{R_N}$.
7. Listen for continue vectors $\boldsymbol{R_N CONT}$. If a continue vector for an alternate service is heard, then return to listening for new vectors.
8. If no continuation message is received after time-out period, then return to listening for new vectors.
9. On receipt of a continue vector perform the local service task. We note that this may be a null task or a task to run a sub workflow as a new initiator or simply to perform an action.
10. On completion of the local service task unbind the received vector again to get $\boldsymbol{Z'_{N+1}}$ and transmit.
11. Listen for responses $\boldsymbol{R_{N+1}}$
12. Arbitrate the responses and issue continue vector $\boldsymbol{R_{N+1}}$
13. Return to listening for new vectors.

7.5. Static workflow control

In the static workflow mode steps 1–12 from the dynamic mode are performed but rather than terminate at step 13 the cognitive layer computes new semantic vectors that are a combination of the current service name $\boldsymbol{Z_s}$ and its position in the unbound vector to essentially create a temporary service name as a parent node or child node—the details of this are given in Section 8.2. These vectors are stored in the memory and the service listens for these vectors. On receipt of a parent or child vector the steps 1–3 and 5–12 are repeated. Step 4 is not required since the name is unique and only this service can respond. In the case of receiving a child vector the layer also accesses the IP address in the associated message and unicasts a 'hello' message to the associated service to create a connection. The cognitive layer than either listens for new requests, since its service can simultaneously be part of multiple workflows, or it waits until the service has completed the current workflow and then resumes listening for new vectors with its original service name.

8. Test cases

In order to demonstrate the applicability and scalability of our encoding scheme, we provide two use cases where we have applied the VSA system to both linear and complex workflows. We also provide an experimental evaluation for the correctness and scalability of the proposed approach.

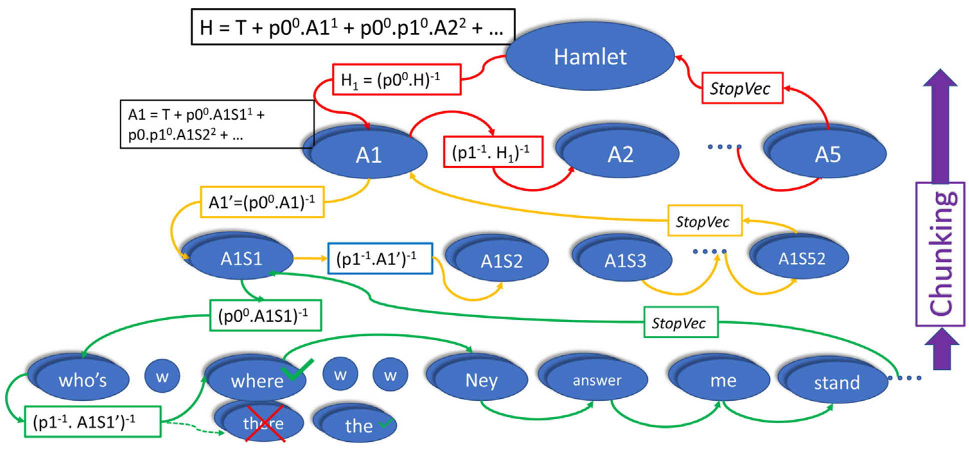

Fig. 3. Hamlet as a serial workflow.

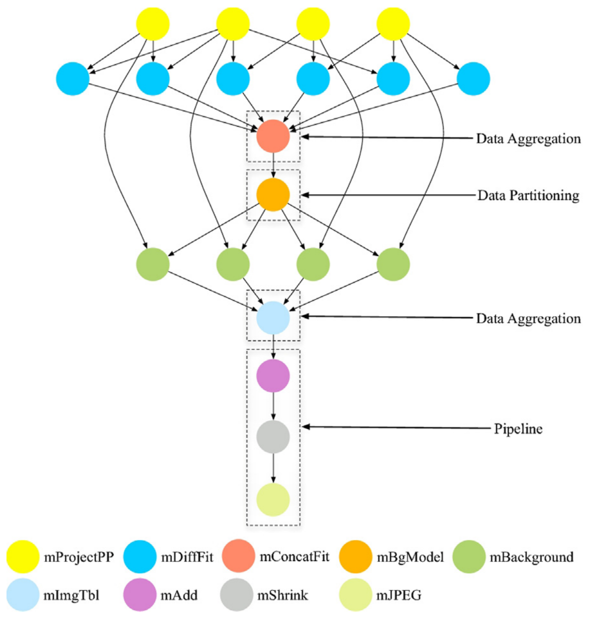

Fig. 4. Montage workflow.

8.1. Dynamic mode linear workflow

To compare with alternative approaches such as those described in [34], we have semantically encoded the entire text of Shakespeare's play Hamlet into this type of hierarchical semantic vector representation. In this example the component services at the lowest level are the 4620 unique words of the play; the semantic level above are the individual stanzas spoken by each character (not shown in the diagram); the level above this are individual scenes of the play (e.g., A1S1, A1S2); next are the five acts, A1–A5 and then finally a single 10 kb vector semantically represents the whole play (Hamlet). A vector alphabet, a unique vector per alphabet character, was used to build compound vectors for each word-service in the play. The idea is that each letter making up a word represents some feature of a service description, i.e., analogous to the different input/output/name/descriptions parts of a real world service. Thus, variable lengths of words and similarity of spellings represent a mix of different services of different complexity and functional compatibility. At the next higher level sentences represent a more complex sub-workflow, and so on. It is important to recognize that the higher level vectors do semantically represent the recursive binding from all the levels below. We use this fact to allow alternative, semantically similar, service compositions to be invoked if the best matching composition is not available. Fig. 3 shows the word service *where* being invoked as an alternate to *there* which was unavailable. Note also, that when *where* completes it automatically re-synchronizes to the original worflow because it simply *unbinds* the next step from the original workflow vector it received.

To simulate QoS matching we employed two random variables representing *current load* and *battery life*. From a requesters point the idea was that *current load* should be minimized and *battery life* should be maximized so that we could try out our min/max idea for encoding QoS as described in Section 4.0.2. Acceptable ranges of values were randomly chosen when encoding service request vectors and each services simulated its own QoS in the same manner. Thus matching on functional as well as QoS criterion was tested. Multiple copies of the individual component vectors at each level are distributed in a communications network as services and by multicasting the top level vector the whole play is performed in a distributed manner with 29,770 component word services being invoked in the correct order.

8.2. Static mode complex workflow

The workflow described in Section 8.1 is a simple linear sequence of services. In this section we describe how the vector representation can be extended to more complex workflows such as those created by the Pegasus workflow generator. Fig. 4 shows a typical Pegasus workflow (the Montage Workflow) having multiple connections between nodes with branching and merging of connections.

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

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

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

Listing 1: DAX snippet

The example workflow in Fig. 4 can be represented as a symbolic vector as follows:

$$\boldsymbol{WP} = \boldsymbol{p0}^0 \cdot (\boldsymbol{Recruit}_{\boldsymbol{Nodes}})^1 + \boldsymbol{p0}^0 \cdot \boldsymbol{p1}^0 \cdot (\boldsymbol{Connect}_{\boldsymbol{Nodes}})^2 + \boldsymbol{p0}^0 \cdot \boldsymbol{p1}^0 \cdot \boldsymbol{p2}^0 \cdot \boldsymbol{Start}^3$$

where:

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

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

Each $\boldsymbol{Z_n}$ in $\boldsymbol{Recruit}_{\boldsymbol{Nodes}}$ is the compound vector representation of each service. In our implementation, the vectors are constructed automatically from the Pegasus DAX file as per Listing 1. The $\boldsymbol{Recruit}_{\boldsymbol{Nodes}}$ vector is built from the <job> entries found in the DAX—we refer the reader to Fig. 4 and Listing 1, where there are 4 mProjectPPs($\boldsymbol{Z_1 s}$), 6 mDiffFitt($\boldsymbol{Z_2 s}$) and so on.

The $\boldsymbol{Connect}_{\boldsymbol{Nodes}}$ vector, built from the <child> entry section of the DAX, defines the producer/consumer relationship between nodes. A node can act as both a parent (producer) and child (consumer) within the workflow, see Fig. 4. Using Listing 1 as an example, the parent, $\mathbb{P}$, and child, $\mathbb{C}$, ends of each edge are constructed as follows:

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

where

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

By binding these three elements together we construct a unique encoding for the parent and child ends of every edge in the DAG. Using Eq. (9) we then represent the edges as an ordered list of parent→child ends, see $\boldsymbol{Connect}_{\boldsymbol{Nodes}}$ above.

8.2.1. Execution of the workflow

The resulting workflow $\boldsymbol{WP}$ is a superposition representing the linear sequence of steps needed to: *(a)* discover the required services, *(b)* connect the selected services together, and *(c)* signal to the selected services that the workflow is composed and work should begin. Therefore, the execution of the workflow proceeds in a similar manner to that described in Section 5.1 but with some additional workflow specific processing carried out by each selected node. The top level vector, *WP* is prepared as per Eq. (11):

$$\boldsymbol{WP}_1 = (p_0^0 \cdot (T + WP))^{-1} = p_0^{-1} \cdot T^{-1} + \boldsymbol{Recruit}_{\boldsymbol{Nodes}} + \boldsymbol{noise}$$

When multicast, this activates the $\boldsymbol{Recruit}_{\boldsymbol{Nodes}}$ service which, operating as a cleanup service, carries out the same operation to initiate the recruitment phase:

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

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

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

It then enters *Listening for Connections Mode* while, as the new *local arbitrator*, it also multicasts the next recruitment request by performing an unbind using, Eq. (13), on its received vector $\boldsymbol{R'_1}$,

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

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

The $\boldsymbol{Recruit}_{\boldsymbol{Nodes}}$ cleanup service detects its stop vector, causing it to perform an unbind, using Eq. (13), and multicast of $\boldsymbol{WP'}$ thereby activating the $\boldsymbol{Connect}_{\boldsymbol{Nodes}}$ phase:

$$\boldsymbol{WP}_2 = (p_1^{-1} \cdot WP_1)^{-1} = p_1^{-1} \cdot p_0^{-2} \cdot T^{-2} + \boldsymbol{Connect}_{\boldsymbol{Nodes}} + \boldsymbol{noise}$$

At this point all recruited services are listening for connection request on their unique parent and child vectors. The activated $\boldsymbol{Connect}_{\boldsymbol{Nodes}}$ service, acting as a cleanup service, uses Eq. (11) to initiate and activate the first *parent* node of the $\boldsymbol{Connect}_{\boldsymbol{Nodes}}$ phase:

$$\begin{aligned}\boldsymbol{Connect}'_{\boldsymbol{Nodes}} &= (p_0^0 \cdot (T + \boldsymbol{Connect}'_{\boldsymbol{Nodes}}))^{-1}\\ \mathbb{P}'_1 &= p_0^{-1} \cdot T^{-1} + \mathbb{P}_1^0 + p_1^{-1} \cdot \mathbb{C}_1^1 + p_1^{-1} \cdot p_2^{-1} \cdot \mathbb{P}_2^2 + \cdots\end{aligned}$$

When a service matches its *parent* vector it performs the next unbind or multicast to activate its associated *child* service, automatically informing the child service of the location of its resources/output/ip-address. When a service receives a multicast matching its *child* vector it can lookup the sender and parent's IP-address and send a unicast *hello* message to the parent, thus establishing the required connection before activating

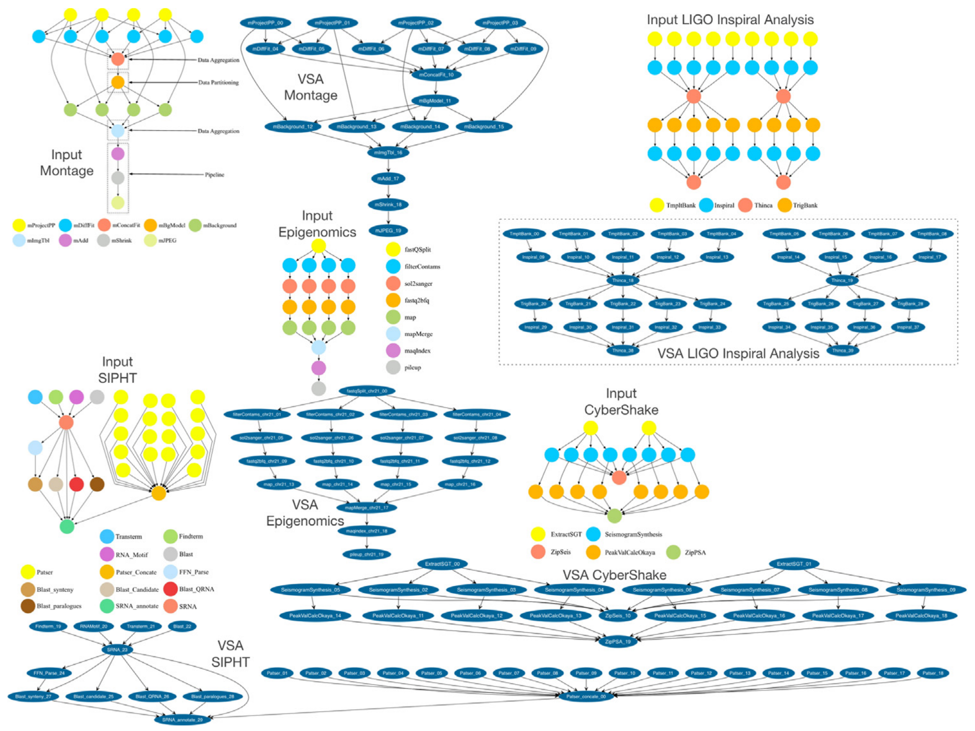

Fig. 5. A comparison of five different DAX workflows as input and the VSA reconstructed workflows from post processing the semantic vector workflow. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the next *parent* by performing a further unbind or multicast of the ***Connect***$_{Nodes}$ vector. This process repeats until the final child request is processed causing the ***Connect***$_{Nodes}$ service to detect its *StopVec* which, in turn, causes it to unbind and multicast the *StartVec* indicating to all nodes that the workflow has been fully constructed and processing can be started.

9. Evaluation

Our evaluation of the scalability of the VSA approach for linear workflows has already been presented using the Hamlet example in Section 8.1. The evaluation was performed using the CORE/EMANE network emulator to simulate a MANET network and used a MANET multicast routing protocol to communicate vectors between the nodes containing the services. Multicasting the top level Hamlet vector results in the whole play being enacted by worker services that generate each word in the play. The VSA workflow implementation of Hamlet has a number of advantages over the Newt [34] implementation. Specifically the Newt implementation requires that the IP address of participating services be known and encoded into the workflow, whereas, our VSA approach can discover the service (word/sentence) needed on the fly using semantic matching. In Newt, if the service specified by IP address becomes unavailable; i.e., we intentionally move it out of wireless range in CORE, then the workflow halts and is broken. In VSA Hamlet, the same action results in the automatic discovery of multiple exact, and near-match candidate word/sentence/services and the best match is then chosen. When multiple functionally equal matches were discovered the *Local Arbitration* function ensured that the service having best simulated utility was chosen and logged as such. The best *'near'* match was chosen when we contrived to make exact matches unavailable in CORE. Additionally, the advantage of passing around the workflow as a vector superposition was highlighted because the stand-in service automatically resynchronized the workflow after *'speaking'* its substitute word by simply performing an unbind and transmit of the workflow vector it received. Newt has none of these capabilities.

Our evaluation of the more complex workflows was aimed at the following: to demonstrate that complex workflows could be automatically encoded into a symbolic vector representation and then recursively decoded to assemble the required work in a decentralized setting; to show that the workflow constructed was also resilient to changes in the communications network; and to demonstrate that services with the highest utility could be identified and selected using the semantic matching mechanism.

For the evaluation, we used five different DAX workflows generated using the Pegasus workflow generator [37]:

1. Montage (NASA/IPAC) stitches multiple input images together to create custom mosaics of the sky.
2. CyberShake (Southern Calfornia Earthquake Center) characterizes earthquake hazards in a region.
3. Epigenomics (USC Epigenome Center and Pegasus) automates various operations in genome sequence processing.
4. Inspiral Analysis (LIGO) generates and analyzes gravitational waveforms from data collected during the coalescing binary systems.
5. SIPHT (Harvard) automates the search for untranslated RNAs (sRNAs) for bacterial replicons in the NCBI database.

We again ran a series of experiments using the CORE/EMANE network emulator to simulate a MANET network. Pegasus DAX workflows were processed using the VSA creator to build the semantic vector workflow encodings and also to generate the service description vectors that semantically describe each of the component services. Multiple copies of the component services were randomly distributed on the network nodes. Each

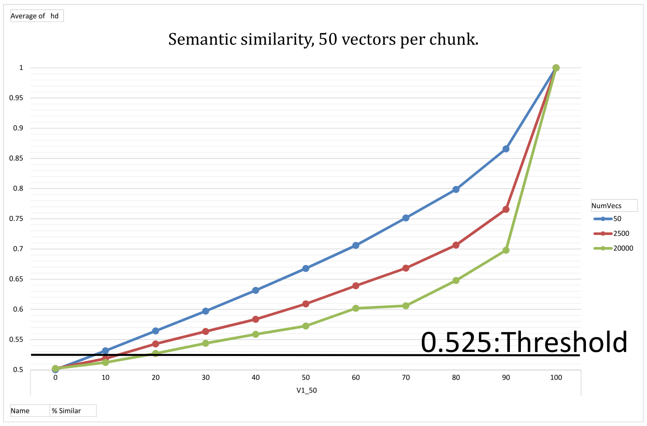

Fig. 6. Semantic similarity 20 k vectors, chunk size 50. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

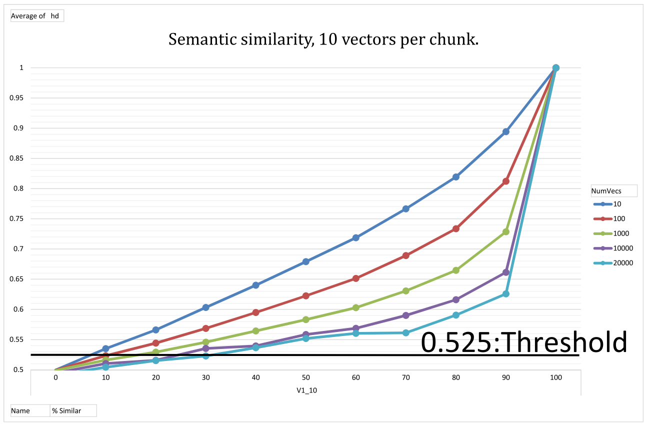

Fig. 7. Semantic similarity 20 k vectors, chunk size 10. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

service was enabled with our VSA cognitive layer containing the appropriate semantic vector for that service. The workflow request vector was launched from some node in the network and the workflow was constructed in a decentralized manner, with control being passed between services as the workflow vector was recursively unbound. During the execution of the process we extracted a range of metrics that provided a detailed log of the run and the order of execution. Using this log we created a graph of the set of nodes and edges that were selected and we used Graphviz to show the result. Fig. 5 shows the results for the five different Pegasus workflows we evaluated. The colored images represent the Pegasus generated workflows and blue workflows show the VSA generated reconstruction of the workflows. Aside from the cosmetic difference, this demonstrates that all workflows were composed and correctly connected accurately in all cases.

To demonstrate the resilience of the approach, we modified the network connectivity to demonstrate that different instances of the correct services were selected and that this still produced the same required workflow. We also demonstrated that if the same services had different QoS utility measures that the services with the higher utility were selected in preference to those of lower utility.

10. Hierarchical VSA scaling preserving semantic similarity

In order to demonstrate that our encoding scheme can scale whilst preserving a measure of semantic similarity, we performed a further empirical evaluation. Using sets of randomly generated vectors we carried out a number of experiments. Two sets, *set1* and *set2*, of 10 kb random vectors where generated and both sets where *chunked* using Eq. (9). The resulting top level vectors were then compared by measuring Hamming distance similarity. The comparison was repeated after randomly choosing an increasing percentage of vectors from set2 and copying them into the same position in set1. The Hamming distance similarity was then recalculated as the sets become increasingly more similar. The expected result was that no similarity would be detected when each set had none or very few common vectors and that Hamming distance similarity would increasingly improve as the sets become similar.

Figs. 6 and 7 show the results using a set size of 20k for chunk sizes of 50 and 10. Each line shows the average similarity – i.e., $(\mathbf{1} - \boldsymbol{HD})$ – of chunks at each level in the chunk hierarchy. For example, in Fig. 7, there is only 1 vector at the top level (green), 8 chunks at each point of red and 400 chunks at each point on the blue line. Comparing Figs. 6 and 7 we see that ability to detect a semantic match decreases for smaller chunk size.

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

11. Conclusions and future work

In this paper, we addressed the complex problem of how to represent and enact decentralized time-critical applications. Specifically we investigated how data analytics tasks, formulated as complex workflows, could operate in dynamic wireless networks, without any central point of control. To this end, we described an architecture that exposes a cognitive layer by using a Vector Symbolic Architecture (VSA) to extend services with semantic service descriptions and time-critical constraints required to specify the QoS/QoE. We demonstrated the viability of this approach by showing an empirical evaluation that such VSA encoding methods work and are scalable. We then described the architecture of our approach and the components it provides to enable decentralized fitness functions for on demand resource discovery and allocation.

We demonstrated that our approach can 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 showed that time-critical QoS and QoE metrics for each workflow, sub-workflow or even service can be encoded into a single vector that provides an extremely compact (10 kb) common workflow format exchange for a MANET, which can be passed around using standard group transport protocols (e.g., multicast). We also showed that semantic comparisons can be made at each level of the architecture to support scoped searching and that the scheme is extensible—i.e., new parameter or constraint can be plugged in and encoded to address practically any real-world scenario.

In the future, we will investigate different schemes for discovery and matchmaking, which are capable of supporting different modes of use. For example, we are currently looking at using the look ahead *peeking* capability of VSAs in combination with proactive announcements that will be capable of pushing utility metrics calculations to the client that need to consume them. A key element of future work is to investigate alternative ways to encode semantics and to measure the semantic similarity of services and their QoS and QoE. We are investigating methods that capture the previous contexts, including QoS metrics, in which a particular workflow has operated in, as well as other methods that avoid rigid ontology style approaches. For time-critical applications in MANET environments we are investigating alternatives to local arbitration that will allow the fittest service to rapidly emerge from a group of compatible competing individual services. Using symbolic vectors to semantically represent services and workflows enables suggested alternative service compositions to be automatically generated when component services of an existing workflow are missing or cannot be accessed. We are investigating if viable alternative compositions can be generated and automatically validated using this approach.

Acknowledgments

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.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.future.2019.04.010.

References

[1] Q.Z. Sheng, X. Qiao, A.V. Vasilakos, C. Szabo, S. Bourne, X. Xu, Web services composition: A decade's overview, Inform. Sci. 280 (2014) 218–238.
[2] S. Corson, J. Macker, Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations, RFC 2501 (Informational), Internet Engineering Task Force, 1999, [Online]. Available: http://www.ietf.org/rfc/rfc2501.txt.
[3] S. Basagni, M. Conti, S. Giordano, I. Stojmenović, in: S. Basagni, et al. (Eds.), Mobile Ad Hoc Networking, IEEE, 2004.
[4] J. Broch, D.A. Maltz, D.B. Johnson, Y.-C. Hu, 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, ACM, New York, NY, USA, 1998, pp. 85–97.
[5] T. Pham, G. Cirincione, A. Swami, G. Pearson, C. Williams, Distributed analytics and information science, in: Information Fusion (Fusion), 2015 18th International Conference on, IEEE, 2015, pp. 245–252.
[6] D. Verma, G. Bent, 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.
[7] T.A. Plate, Distributed Representations and Nested Compositional Structure, University of Toronto, Department of Computer Science, 1994.
[8] R.W. Gayler, Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience, 2004, arXiv preprint arXiv:cs/0412059.
[9] P. Kanerva, Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors, Cogn. Comput. 1 (2) (2009) 139–159, [Online]. Available: http://dblp.uni-trier.de/db/journals/cogcom/cogcom1.html#Kanerva09.
[10] D. Kleyko, Pattern Recognition with Vector Symbolic Architectures (Ph.D. disseration), Luleå tekniska universitet, 2016.
[11] G.E. Hinton, Mapping part-whole hierarchies into connectionist networks, Artificial Intelligence 46 (1–2) (1990) 47–75.
[12] C. Eliasmith, T.C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, D. Rasmussen, A large-scale model of the functioning brain, Science 338 (6111) (2012) 1202–1205,.
[13] M.N. Jones, D.J.K. Mewhort, Representing word meaning and order information in a composite holographic lexicon, Psychol. Rev. 114 (1) (2007) 1–37.
[14] G.E. Cox, G. Kachergis, G. Recchia, M.N. Jones, Toward a scalable holographic word-form representation, Behav. Res. Methods 43 (3) (2011) 602–615.
[15] G. Recchia, M. Sahlgren, P. Kanerva, M.N. Jones, Encoding sequential information in semantic space models: comparing holographic reduced representation and random permutation, Comput. Intell. Neurosci. 2015 (2015) 58.
[16] T.A. Plate, Holographic Reduced Representation: Distributed Representation for Cognitive Structures, CSLI Publications, Stanford, CA, USA, 2003.
[17] C. Simpkin, I. Taylor, G.A. Bent, G. de Mel, R.K. Ganti, A Scalable Vector Symbolic Architecture Approach for Decentralized Workflows.
[18] A. Van Moorsel, Metrics for the internet age: Quality of experience and quality of business, in: Fifth International Workshop on Performability Modeling of Computer and Communication Systems, Arbeitsberichte des Instituts für Informatik, Universität Erlangen-Nürnberg, Germany, 34, (13) Citeseer, 2001, pp. 26–31.
[19] Y. Liu, A.H. Ngu, L.Z. Zeng, Qos computation and policing in dynamic web service selection, in: Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters, ACM, 2004, pp. 66–73.
[20] E. Vinek, F.T.A. Viegas, et al., ATLAS Collaboration, Composing distributed services for selection and retrieval of event data in the atlas experiment, J. Phys.: Conf. Ser. 331 (4) (2011) 042027.
[21] E. Vinek, P.P. Beran, E. Schikuta, A dynamic multi-objective optimization framework for selecting distributed deployments in a heterogeneous environment, Procedia Comput. Sci. 4 (2011) 166–175.
[22] J. Sterle, M. Rugelj, U. Sedlar, L. Korvsivc, A. Kos, P. Zidar, M. Volk, S. Toral, A novel approach to building a heterogeneous emergency response communication system, Int. J. Distributed Sens. Netw. 2015 (2015).
[23] E.D. Nitto, M.A.A. d. Silva, D. Ardagna, G. Casale, C.D. Craciun, N. Ferry, V. Muntes, A. Solberg, Supporting the development and operation of multi-cloud applications: the MODAClouds approach, in: 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2013, pp. 417–423.
[24] M. Wieczorek, R. Prodan, T. Fahringer, Scheduling of scientific workflows in the askalon grid environment., SIGMOD Rec. 34 (3) (2005) 56–62.
[25] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wieczorek, ASKALON: A development and grid computing environment for scientific workflows, in: Workflows for e-Science, Springer, New York, 2007, pp. 143–166.
[26] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, 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.
[27] P. Kacsuk, P-grade portal family for grid infrastructures, Concurr. Comput. : Pract. Exper. 23 (2011) 235–245, [Online]. Available: http://dx.doi.org/10.1002/cpe.1654.
[28] 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, D. Katz, Pegasus: a framework for mapping complex scientific workflows onto distributed systems, Sci. Program. J. 13 (3) (2005) 219–237.
[29] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, P. Li, Taverna: A tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054.
[30] A. Harrison, I. Taylor, I. Wang, M. Shields, WS-RF workflow in Triana, Int. J. High Perform. Comput. Appl. 22 (3) (2008) 268–283,.
[31] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, Y. Simmhan, The trident scientific workflow workbench, in: Proceedings of the 2008 Fourth IEEE International Conference on eScience, IEEE Computer Society, Washington, DC, USA, 2008, pp. 317–318, [Online]. Available: http://dl.acm.org/citation.cfm?id=1488725.1488936.
[32] T. Glatard, J. Montagnat, D. Lingrand, X. Pennec, Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR, Int. J. High Perform. Comput. Appl. 22 (2008) 347–360, [Online]. Available: http://dl.acm.org/citation.cfm?id=1400050.1400057.

[33] B. Balis, Increasing scientific workflow programming productivity with hyperflow, in: Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science, WORKS '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 59–69, [Online]. Available: http://dx.doi.org/10.1109/WORKS.2014.10.
[34] J.P. Macker, I. Taylor, Orchestration and analysis of decentralized workflows within heterogeneous networking infrastructures, Future Gener. Comput. Syst. (2017).
[35] G. Bent, P. Dantressangle, D. Vyvyan, A. Mowshowitz, V. Mitsou, A dynamic distributed federated database, in: Proc. 2nd Ann. Conf. International Technology Alliance, 2008.
[36] G. Bent, P. Dantressangle, P. Stone, D. Vyvyan, A. Mowshowitz, Experimental evaluation of the performance and scalability of a dynamic distributed federated database, in: Proc. 3rd Ann. Conf. International Technology Alliance, 2009.
[37] Workflow Generator Pegasus, https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.
[38] Graphviz — Graph Visualization Software, http://www.graphviz.org/Home.php.

Chris Simpkin has worked as a design engineer on high speed control systems including design of analogue control loops, dedicated digital computer control systems. Chris worked for IBM for 10 years in various areas including stress testing of IBM's flagship S/390 G5 Parallel main frames, IBM CICS and IBM Message Queuing products. In 1998 Chris qualified as an Optometrist and spent 10 years managing an Optometry business. Chris is currently doing a PhD at Cardiff University, UK, focusing in the use of machine learning algorithms for distributed data analytics applications.

Ian Taylor is a Professor at the University of Notre Dame and a Reader at Cardiff University. He has a degree in Computing Science, a Ph.D. studying neural networks applied to musical pitch and he designed/implemented the data acquisition system and Triana workflow system for the GEO600 gravitational wave project. He now specializes in Blockchain, open data access, Web dashboards/APIs and workflows. Ian has published over 180 papers (h-index 41), 3 books and has won the Naval Research Lab best paper award in 2010, 2011 and 2015. Ian acts as general chair for the WORKS Workflow workshop yearly at Supercomputing.

Graham Bent was formally an IBM Senior Technical Staff Member and Master Inventor. He retired from IBM in January 2016. He now works for IBM Research as a contractor. Over the past 10 years Graham has been undertaking research on large scale distributed databases; new encryption techniques for distributed secure computing using fully homomorphic encryption. He is currently involved in a new International Technology Alliance program on Distributed Analytics and Information Science (DAIS ITA). His current research is in the development of intelligent agents for distributed analytics using brain inspired neuromorphic computation.

Geeth de Mel is a Research Staff Member with IBM Research and based at Hartree Centre, UK. His research interests are in applying artificial intelligence – especially in Semantic Web technologies – techniques for decision support systems in the presence of (or lack of) dynamicity, trust, and provenance. He graduated from the University of Aberdeen, Scotland and did his postdoctoral work at the US Army Research Laboratory till he joined IBM Research TJ Watson Research Center, Yorktown, USA in June 2013. His current research focus is on data semantics to better support information correlations and corroboration, and techniques to reformulate questions based on pragmatics.

Swati Rallapalli is a research staff member at the IBM T.J. Watson Research Center. Her work is focused on distributed video analytics (using CNNs/RNNs) on mobile systems and user analytics. In particular, her research focuses on inferring user activities from multiple mobile sensors under stringent resource constraints. She also serves as a principal investigator on the Quality Aware Semantic Video Analytics under US Army Research Lab funded Network Science Collaborative Technology Alliance. She received her PhD from Univ of Texas, Austin in 2014.

Liang Ma is a Research Staff Member with the Department of Cognitive Distributed Systems, IBM T. J. Watson Research Center, NY, USA. He was a recipient of the International Conference on Distributed Computing System Best Paper Award, the IBM Patent Award 2013, the Best Student Paper Award of ITA in Network & Information Sciences 2013, the ACM Internet Measurement Conference Best Paper Award Nomination, the INFOCOM 2014 Student Travel Grant, and the winner of Outstanding Graduate Student 2008 and Excellent Student Awards four times from 2003 to 2006. He received his PhD from the Imperial College, London in 2014.

Mudhakar Srivatsa is a distinguished research staff member at the IBM T.J. Watson Research Center. He served as the Industry Technical Area Leader (2011–16) for Secure Hybrid Networks on the US/UK Government funded Network and Information Sciences International Technology Alliance. His work spans cognitive analysis of spatiotemporal data gathered from IoT sensors for distributed activity detection and robustness under adversarial settings such as quantitative time and graphical side channels. He is an IBM master inventor, authored over 100 research papers, 28 granted US patents, and received three IBM outstanding technical achievement awards.