Experiments with a Machine-centric Approach to Realise Distributed Emergent Software Systems

Roberto Rodrigues Filho
School of Computing and Communications
Lancaster University
Lancaster, UK
r.rodriguesfilho@lancaster.ac.uk

Barry Porter
School of Computing and Communications
Lancaster University
Lancaster, UK
b.f.porter@lancaster.ac.uk

ABSTRACT

Modern distributed systems are exposed to constant changes in their operating environment, leading to high uncertainty. Self-adaptive and self-organising approaches have become a popular solution for runtime reactivity to this uncertainty. However, these approaches use predefined, expertly-crafted policies or models, constructed at design-time, to guide system (re)configuration. They are *human-centric*, making modelling or policy-writing difficult to scale to increasingly complex systems; and are *inflexible* in their ability to deal with the unexpected at runtime (e.g. conditions not captured in a policy). We argue for a *machine-centric* approach to this problem, in which the desired behaviour is autonomously learned and emerges at runtime from a large pool of small alternative components, as a continuous reaction to the observed behaviour of the software and the characteristics of its operating environment. We demonstrate our principles in the context of data-centre software, showing that our approach is able to autonomously coordinate a distributed infrastructure composed of emergent web servers and a load balancer. Our initial results validate our approach, showing autonomous convergence on an optimal configuration, and also highlight the open challenges in providing fully machine-led distributed emergent software systems.

CCS Concepts

•**Software and its engineering** → **Distributed systems organizing principles;** *Software design engineering;*

Keywords

Self-adaptive systems, Self-organising systems, Distributed emergent software systems

1. INTRODUCTION

Current systems are reaching a level of complexity that is beyond human capacity to fully understand and manage. This complexity increases when considering large-scale distributed systems that support modern applications, as is the case of widely used Internet services running in data centres.

ARM 2016, December 12-16, 2016, Trento, Italy

 ISBN 978-1-4503-4662-7/16/12…$15.00

DOI: http://dx.doi.org/10.1145/3008167.3008168

Self-organising systems [10] are one solution to handle this complexity in environments with constant change. Such systems are able to coordinate and execute distributed architectural adaptation by adjusting their infrastructure and behaviour to accommodate changes in their environment.

Current self-organising techniques use *policies* to guide system adaptation (e.g. [2, 5, 9]). Human-defined policies express the context in which adaptation should be executed and how adaptations should occur, i.e., to what behaviour the system should adapt. These are *human-centric* approaches which have limited impact on the complexity issue, as developers must retain a detailed understanding of the system and its behaviour in each environment it may encounter. In addition, these approaches rely on *prediction* of how a system will respond to environments (which may turn out to be false) and result in *inflexibility* to operating environment ranges that were not considered at design time.

We consider *distributed emergent software systems*, which use machine learning to emerge into optimal designs from a pool of available components, based on real-time observations of their current environment and their performance. In contrast to the above, this is a *machine-centric* approach which moves the burden of complexity into software itself, avoids the need for prediction of behaviours, and supports total flexibility to the actual environments encountered as they are observed and learned about. Our approach is based on our existing work on local emergent software, which we have demonstrated can assemble, reason over and design software at runtime using limited information [8, 7].

Our key contributions in this paper are: (i) an analysis of the design space of distributed emergent software systems, based on our experience of implementing them to date; (ii) a prototype design of our distributed emergent software framework and case study application for which we examine distributed emergent behaviour; and (iii) an initial experimental evaluation, which highlights the major benefits and challenges of this approach. As our case study application in this paper we examine two web server nodes along with a load balancer. Each of these nodes exhibits locally emergent properties, able to be composed from a set of possible components, and must form a globally optimal solution in a given operating environment. Our results demonstrate that different operating environment conditions exist for this system, which drive the need for different compositions of components; that our framework can autonomously locate the optimal compositions with very little prior information; and that different approaches to distributed coordination significantly impact the behaviour of the emergent system.

The rest of this paper is structured as follows: in Sec. 2 we discuss the design space of distributed emergent software, and in Sec. 3 we present our prototype implementation. In Sec. 4 we then present our initial experimental work, and we discuss related work in Sec. 5. We conclude in Sec. 6.

2. DESIGN SPACE

In this section we define our model of distributed emergent software systems and discuss the key challenges in their construction. First, however, we summarise the concept of local emergent software systems [8] as necessary background.

2.1 Local Emergent Software Systems

As detailed in [8], local emergent systems are defined as systems whose behaviours emerge at runtime to satisfy a goal in the face of fluctuations in their current operating environment. We formalise this with the following model:

There exists a goal G, expressed in a given form. A set of small software units SU exists that can be composed together into systems that achieve this goal, where each $u \in SU$ has one or more behavioural *variations* (implementations that offer the same functionality using different techniques). One or more u emits a stream of 'metrics' describing the current health of u, and one or more u emits a stream of 'events' describing the software's current external stimuli (i.e. received inputs, or deployment environment characteristics). The aim of an emergent software system is then to continually maximise its satisfaction of G by assembling the most optimal collection of u (where 'satisfaction' is the combined health of all chosen u as reported by metrics) in each set of deployment environment conditions experienced by the software at runtime (as characterised by events).

This model assumes that at least some of the software components involved in the construction of a target system have a natural set of *behavioural variations*; this may be different sorting algorithm implementations, different cache replacement strategies, or solutions that do or do not use a memory caching component to deliver a service. We also assume the property of 'divergent optimality', such that some compositions of component variants are most efficient in some operating environments, while other compositions are most efficient in other operating environments. In [8] and [7] we demonstrate these properties for a single web server instance. We do not elaborate on this further here, but the interested reader is referred to [8] for more detail.

2.2 Distributed Emergent Software Systems

In the distributed case, involving multiple collaborating nodes, we augment our above model as follows:

There exists a set of nodes N, each with its own local goal G, such that some of those goals depend on their node interacting with other nodes within N. Each $n \in N$ is a complete emergent software system as described above. The set of local goals may either be identical (for example in a pure peer-to-peer system) or may be different across some or all nodes (for example in a hierarchical system).

This creates a new problem, beyond the local case, of how to successfully converge on a distributed (or 'global') emergent software system that is (indirectly) built from a large pool of available software components on a set of different nodes. Where assembly of a single emergent node can easily be 'centrally' controlled, for example, the latencies involved in networking can make this difficult in a distributed setting.

We note that, in our current implementation, we do not attempt to model network connections as first-class aspects of our model. Instead we leave all routing decisions to the application itself, with the exception that selecting different components on individual nodes may in itself affect the way that network routing or peering takes place. We now discuss the major challenges here in more detail.

2.2.1 Combinatorial explosion in search space

For a local node, the number of possible ways in which a system can be composed already grows super-linearly with more component variants. This is because the number of ways in which components can be *combined* grows quickly with the number of available components and their variants.

This creates a very large search space for a machine learning algorithm to explore until it finds an optimal composition for any given set of operating environment conditions. This search space size problem becomes combinatorial in a distributed setting: consider a web server with 50 valid compositions of components, and a load balancer with 10 valid compositions. With one web server and one load balancer working together, this creates 500 total permutations of the combined system. If we add another web server, this total increases to 25,000, and so on. If the web server instances are themselves reliant on a third tier of emergent software (such as database nodes), this problem grows exponentially.

2.2.2 Locus and personality of control

A distributed emergent software system can be controlled from a single central point with a global view of the system, or can be controlled by individual control entities on each node. To generalise this, arbitrary groupings of nodes can be controlled together as appropriate (i.e. single nodes, clusters of two, three or four nodes, etc., up to all nodes being treated as a single group for emergent software control). Orthogonally to this, the control system can adopt different 'personality' types for each node: a controller can, for example, act in an entirely selfish way, maximising the satisfaction degree of its local control scope, or in an altruistic way, attempting to maximise the satisfaction degree of both themselves and their connected control scopes.

2.2.3 Information sharing

To help overcome the combinatorial search space explosion, emergent software instances can share information in various ways. This includes the fact that they are currently in an 'exploration' or 'exploitation' mode of learning, indicating their relative behavioural consistency to nodes that interact with them; sharing the metrics and events that are observed at a given node with other nodes around them; or sharing learned information (such as the optimal composition for a given set of operating environment conditions) with similar nodes to avoid re-learning of the same information at multiple locations in a distributed system. These levels of sharing may best be represented by a unified information bus tied in to the learning models on each node.

2.2.4 Interference effects

It is likely that changes to the composition of any one node affect the way that other nodes experience the world. This may be evident for both metrics and events. In metrics, the metric data perceived at a given 'dependent' node are likely affected by the behaviours of other nodes with which it

interacts in the distributed system, such that changes to the composition of emergent software at those other nodes may cause unexpected changes in metrics of the dependent node. In events, differences in components such as the scheduling module of a load balancer can change the request pattern observed at each server in a cluster – in this case the load balancer impacts the environment observed by web servers. These complex interference effects between nodes can in turn make distributed real-time learning difficult.

2.2.5 Behavioural mismatches

Finally, distributed emergent software systems must avoid behavioural mismatches at different nodes – for example the use of different encoding schemes or encryption algorithms that make nodes incompatible. While these factors can be trivially captured by rules or constraints in a human-centric approach to distributed system adaptation, other mechanisms are needed in machine-centric emergent software, such that freedom of exploration and learning is not impeded while simultaneously assuring global system validity.

3. DISTRIBUTED EMERGENT SOFTWARE FRAMEWORK

In this section we present a fully functional framework to realise distributed emergent software systems. This is built on our local emergent software framework (described in [8]). All of our source code for this paper is available online at [1]. This work was carried out using the ultra-adaptive *Dana* component-based programming language [6], though other component models such as OSGi could potentially be used.

3.1 Local Framework

Our local emergent software framework is composed of three main modules: **Assembly**, **Perception** and **Learning**. The Assembly module discovers all available components for use in the target system, by starting from a 'main' component and recursively inspecting required interfaces to search for all components offering compatible provided interfaces. A list of all possible configurations of the target system is then generated, any of which the assembly module can (re)assemble the target system into at runtime. The Perception module allows software components to generate numerical information (e.g., response time in *ms*) about the system's perceived operating environment and its current health condition. The Learning module uses both Assembly and Perception to experiment with available compositions of components and collect perception data generated by any in-use components. The learning algorithm we use for this paper in particular is a simple reinforcement learning approach. This is chosen to establish an initial baseline for future comparison, when more elaborate or complex learning approaches are used. Our simple reinforcement learning approach uses an *exploration* and *exploitation* phase. During exploration, each available software composition is tested for a fixed observation window time, and perception data collected after each test. Following this, the operating environment that existed during that exploration phase is quantified and the best-performing configuration for that environment noted; the exploitation phase then selects that best configuration for use, until perception data indicates that either the environment of system performance deviates outside the current window. For more details, see [8].

3.2 Distributed Framework

In our local framework the Assembly, Perception and Learning modules interact by local function calls. To help realise the distributed framework we define a RESTful web services API to allow remote interaction among the framework modules. The Assembly module offers the functions `setMain()`, `getConfigs()` and `setConfig()` functions. The Perception module provides the function `getPerception()`.

The `setMain()` function is used to discover all possible components and compositions of components for the target system, starting from a main component. The `getConfigs()` function returns a list of possible valid configurations of the assembled system, and `setConfig()` changes the system's configuration. The `getPerception()` function returns numeric information generated by the system's components.

In the distributed framework, the Assembly and Perception modules are grouped to form a **Controller** module. The Learning module must register itself with a Controller by calling a `registerLearner()` function in order to gain access to the Controller's API. This process ensures that only one Learning module is allowed to access the Controller at a time, guaranteeing consistency in the learning process across multiple independent Learning modules.

We also define a RESTful web services API for the Learning module itself to enabling learning coordination, with the functions `on()`, `off()` and `getKnowledge()`. The `on()` function activates the Learner and takes a list of Controller endpoints (IP addresses) that the Learning module will control. The `getKnowledge()` function can be used among Learners to share learned information for cooperative distributed learning strategies. For a fully centralised learning strategy, only one Learning module is activated and controls all nodes in the system. For a fully decentralised strategy, one Learning module is activated for each node in the system.

3.3 Target Software System

As a case study distributed system for this paper we use a web server (of which there may be many instances) and a load balancer which acts as a rendezvous point between clients and a cluster of web servers.

3.3.1 Web Server

Our web server uses three main concepts: a *HTTPHandler*, *Cache* and *Compression*. The *HTTPHandler* represents the core functionality of the web server, defining how HTTP requests are handled. Different implementations of *HTTPHandler* process requests differently by combining extra elements such as *Cache* and *Compression*. The Cache interface defines the general caching functionality, for which there are various implementations. Similarly, the Compression interface defines the general compression functionality, with a set of available implementations. These elements form the basis of our web server, allowing the formation of four different architectural 'groups' which serve requests by: i) simply reading each requested resource direct from disk; ii) caching a limited set of requested resources in memory, iii) compressing requested resources to send as response, iv) *caching* a limited number of compressed resources. For this paper we employ just one example caching implementation and one compression implementation, yielding four possible configurations (though we have many more available). For perception data, the web server emits response times as metrics, and requested content types and volumes as events.

3.3.2 Load Balancer

Our load balancer receives HTTP requests from clients, forwards them to a specific web server using a scheduling algorithm, and returns the response to the client. Its main functionality is a *RequestHandler*, for which we have implemented four scheduling variants: i) Round Robin, ii) Mime-type, iii) Cache-Location, and iv) an approach that enables the load balancer to locally cache the most recently requested files, using a Round Robin scheduling algorithm. The Round Robin variant simply has a (circular) list of available web servers and forwards each new client request to the next server on the list. The Mime-type variant considers the MIME type of each HTTP request and builds a mapping of MIME types to web servers, such that for example all image requests are forwarded to server A, all text requests to server B, etc. Finally, the Cache-Location variant remembers the web server to which a request for a given resource was last sent (if any) and forwards further requests for that resource to the same server. The load balancer emits the same perception data as our web server (response times as metrics and request content types and volumes as events).

4. EVALUATION

In this section we present results of our initial experimental work with our distributed emergent software framework, using our load balancer / web server system. This gives key insights into some of the major challenges in building systems in this way. The main results that we present demonstrate that (i) different optimal compositions of our example system exist under different environments; (ii) these optimal compositions can be autonomously discovered by our framework; and (iii) coordinated and decentralised approaches in the learning dimension provide significantly different overall behaviours in our emergent software system.

All of our experiments were performed with our two web servers running on two identical rackmount servers in a typical datacentre environment, with our load balancer on a third machine, and our client (generating workloads) residing on a fourth machine in a different subnet. All of our source code for this paper, with instructions on how to reproduce our experiments, is available online at [1].

4.1 Divergent optimality

Our results here, shown in Fig. 1 and Fig. 2, indicate that: i) different global distributed system configurations behave very differently in the same operating environment conditions; ii) for two different environments, there are notably different global system configurations that perform optimally; and iii) there are very clear *groups* of configurations with similar performance levels in both environments.

In detail, Fig. 1 and Fig. 2 show the average response time to client requests, as reported at the load balancer, for every possible global distributed system configuration (i.e. every possible configuration of components for every local node, combined) for two tested environments, characterised by different workloads (Workload 1 and Workload 2).

For Workload 1, shown in Fig. 1, there is a difference of 193 ms in request handling latency between the best and worst performing distributed configurations. Similarly, for Workload 2 shown in Fig. 2, the difference in request handling latency between the best and worst architecture is 126 ms. This clearly shows that different configurations have significant impact on overall performance (result (i)).

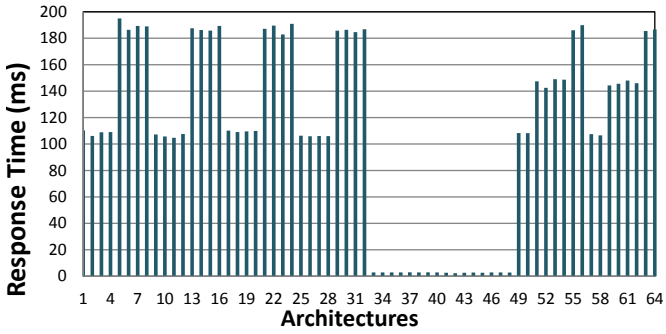

Figure 1: Average response time of every available distributed configuration when using Workload 1.

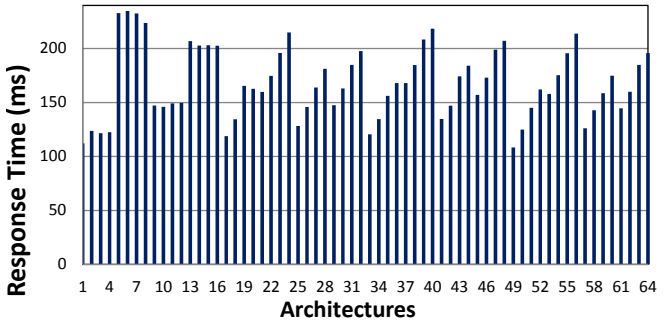

Figure 2: Average response time of every available distributed configuration when using Workload 2.

Comparing these two graphs, it is also noticeable that the best performing global architecture in both scenarios are different (result (ii)). For Workload 1, the best architecture is #33 (LB-C, WS1-GZ, WS2-GZ)[1], while for Workload 2 the best architecture is #49 (LB-RR, WS1-GZ, WS2-GZ).

Considering that Workload 1 consists of one client repeatedly requesting only one text-only html file, global architecture #33 performs best because, in this configuration, the web servers always compress the requested files, and once the file is returned to the load balancer, it is stored in a small content cache at the load balancer. Thereafter, all subsequent requests for the same file can be retrieved instantly by the load balancer itself, from its local cache, avoiding the communication between load balancer and web servers, and therefore significantly reducing response time.

On the other hand, for Workload 2, which consists of one client repeatedly requesting a different text-only html file for every request, architectures with caching will not perform well due to the frequency of cache misses. As a result, the best performing architecture is one which does not use caching at either the load balancer or the web servers. The architecture #49 defines a round-robin scheduling algorithm for the load balance, and web servers that return compressed files (from disk) as responses. As for the load balancer scheduling algorithm (LB-RR), each incoming request will be evenly passed over to the web servers.

The last notable result here is the equivalent performance of large groups of architectural configurations (result (iii)). This is easily identified in both workloads, though is visually

[1]LB stands for load balancer, WS1 stands for web server 1, WS2 stands for web server 2. The configurations are indicated by their initials: C for Cache, GZ for Compression, CGZ for Cache and Compression and, finally, RR for Round Robin.

more obvious in Fig. 1. One clear reason for this is that both web servers are running on machines with the exact same hardware features and capacity. Thus, a global distributed configuration that sets WS1 to configuration X and WS2 to configuration Y is essentially the same (performance-wise) as having WS1 set to Y and WS2 to X. Furthermore, whenever requests are limited to a subset of the system's nodes, all configurations of the unreachable nodes do not affect the system's performance, making those configurations indifferent for the system. This situation is observed in two cases: the first case happens when the load balancer, due to its scheduling algorithm, forwards 100% of the incoming requests to only one web server. This is observed in architectures #1 to #32 in Workload 1, and from #1 to #16 in Workload 2. The second case happens when the load balancer forwards its incoming requests only once to one web server, and all subsequent requests are handled by the load balancer itself due to its local caching configuration. This happens only in Workload 1 and is observable for architectures #33 to #48.

4.2 Learning behaviour

The above results provide a ground truth, informing us which distributed system configurations are the best options for our two workloads. We now examine different learning strategies, using our framework, which autonomously discover this at runtime. In all cases we use a simple reinforcement learning approach, but we examine this approach in a global configuration, where a single instance of the learning algorithm controls the entire system, and also in a local configuration, where every individual node learns in isolation.

The results are shown in Fig. 3 and Fig. 4. Both graphs show the learning process and its convergence to the optimal configuration. On both graphs we show the performance of three different configurations of our system, for comparison: the learning line is the version running our distributed emergent software framework to control the system (the blue and purple lines in Fig. 3 and Fig. 4 respectively), whereas the red and green lines are a fixed architectural configuration that we use as reference points (i.e. their configurations do not change over time). The configuration represented by the red line is LB-C, WS1-GZ and WS2-GZ, and is the best performing distributed configuration for Workload 1, as shown by our earlier results in which we manually tested each configuration. The green line then represents the architectural configuration LB-RR, WS1-GZ and WS2-GZ, which is the best performing configuration for Workload 2. At the midpoint of both experiments we change the workload from Workload 1 to Workload 2, therefore demonstrating the way in which our framework learns the change in environment.

These graphs show that: i) the centralised learning approach, in both workloads, autonomously identifies the optimal global architectural configuration, with no prior information; ii) the decentralised learning approach, for Workload 1, converges faster than the coordinated learning approach, but it never converges for Workload 2.

In detail, the centralised learning approach, as shown in Fig. 3, takes 320 seconds (∼5 minutes) to find the optimal configuration for each environment it encounters. This is because our learning algorithm works by exposing each of the 64 available configurations and observing them for 5 seconds, after which it selects the configuration that had the lowest average response time. The advantage of this approach is that it will always find the optimal solution, due to its exhaustive search. On the other hand, this approach does not scale to larger numbers of components and/or nodes due to the combinatorial explosion that occurs when doing this.

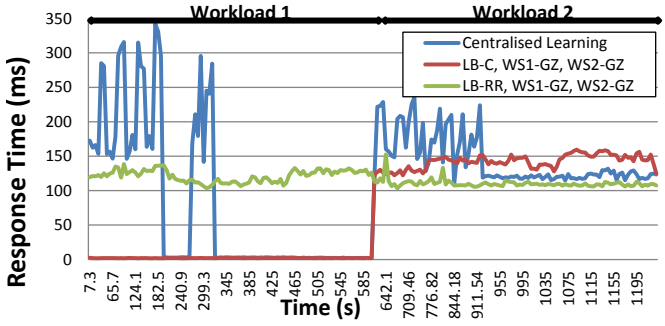

Figure 3: Performance of our coordinated learning approach for two different workloads, compared with static baseline configurations. The spikes at the beginning of the coordinated learning curve for both workloads represent the exploration phase.

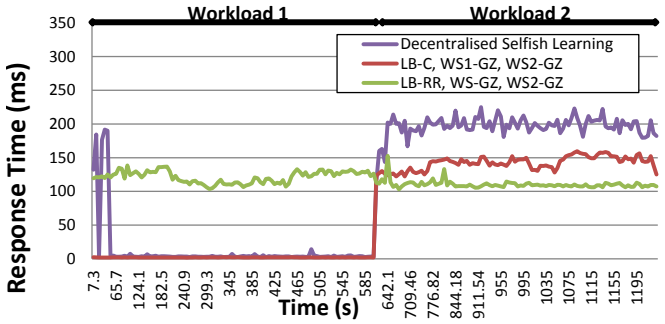

Figure 4: Performance of our decentralised learning approach for two different workloads, compared with static baseline configurations. The spikes at the beginning of the selfish learning curve for Workload 1 represent the exploration phase.

The decentralised (selfish) learning approach, by comparison, converges 15 times faster than the centralised approach when exposed to Workload 1, as shown in Fig. 4. The rapid convergence here is because the local learning process at the load balancer must only iterate through 4 different configurations, quickly identifying the local content caching in the load balancer as the best option, this configuration suffers little performance impact from the web servers (as was seen in Fig. 1). However, we see that this approach never converges on the optimal solution when exposed to Workload 2. In this more complex workload, in which the 'local content caching' solution at the load balancer is not the best option, this lack of convergence is due to a complex set of interference conditions: first, because each local emergent system is exploring independently, it is unlikely that all nodes will simultaneously happen to be in the globally optimal set of configurations at the same moment; second, the locally optimal solution for the web server nodes is not necessarily the globally optimal one; and third, the act of exploring different configurations on different nodes causes constant changes in observed metrics and observed events at both the load balancer and the web servers. In the more complex Workload 2, these conditions mean that all nodes in the system are effectively in a constant state of re-learning.

Finally, we note that our workloads for this paper were synthetically generated in order to carefully control our tests; in future work we will also examine real-life workloads.

5. RELATED WORK

This section surveys the most relevant work in self-adaptive and self-organising systems, presenting different ways for enabling software adaptation and optimisation at runtime.

Static adaptation policies are the most common approach to enable software re(configuration). This approach requires a predefined correlation of operating conditions and software configurations. Examples of this include the use of manually-written temporal rules to specify and constraint software adaptation [5], or encoding adaptation rules in a causally-connected model of the running system via **models@runtime** to manage multi-cloud applications [2]. These approaches rely on a set of rules (sometimes realised within a model of the software system) defined by experts in advance. Our approach goes a step further that this by self-assembling software and automatically learning the creation of rules by which adaptation is needed based on observed conditions.

Dynamic policies approaches improve on static adaptation rules by enabling updates to the policies as new conditions arise. Elkhodary *et al.* propose a feature model framework to enable self-adaptive architectures [4]. Offline-learning is used to create functions that supports system's adaptation, and online-learning is used to tune the functions when new environment conditions are detected. Our approach differs from by placing the learning process in the system, starting with no prior information, and allowing active experimentation with the available software configurations under the conditions actually experienced at runtime.

Biologically-inspired approaches are commonly used to implement adaptive solutions by incorporating patterns of behaviours found in nature to help optimise software systems. There are multiple examples of how ant colony behaviour can be used in the context of optimisation problems [3], including routing algorithms, scheduling algorithms, and so on. This class of solution consists of encoding in algorithms ants' ability to find the shortest path to food using pheromone trails. These approaches generally involve software developers writing problem-specific algorithms inspired by nature; in contrast our work seeks a more general approach to autonomous system assembly and optimisation.

Coordination is often required in distributed adaptive systems. A recent example describes an approach to allow structural adaptation of decentralised coordination processes [9], introducing a voting scheme to solve conflicts between adaptive elements on the system. Though the overall approach to guide software adaptation relies heavily on predefined policies, the decentralised coordination of systems adaptation provides insightful concepts that may be complementary to our work, particularly in the exploration of hybrid learning solutions based on information sharing.

6. CONCLUSION

In this paper we introduce the first example of a machine-centric approach to realise distributed emergent software systems. We evaluated a prototype framework operating in two learning modes (centralised and decentralised) in the context of distributed system composed of two instances of a web server and one load balancer, each with four distinct architectural configurations. Our results show that there are different optimal global configurations available even in a very simple distributed system, when that system is broken down into small components with variation. We also demonstrate that a coordinated learning approach can autonomously locate the optimal distributed emergent software system across multiple nodes. However, this solution does not scale to larger numbers of nodes. A fully decentralised approach, meanwhile, does scale because it does not suffer from the combinatorial explosion problem, but does not converge on an optimal global solution. In future work we intend to examine hybrid solutions that offer the best of both of these alternatives to learning.

7. ACKNOWLEDGEMENTS

This work was partly supported by the UK EPSRC in the *Deep Online Cognition* project, grant no. EP/M029603/1. Roberto Rodrigues Filho would like to thank his sponsor, CAPES Brazil, for the scholarship grant BEX 13292/13-7.

8. REFERENCES

[1] Demos and code from this paper with instructions: http://research.projectdana.com/arm2016rodrigues.

[2] L. Cianciaruso, F. Di Forenza, E. Di Nitto, M. Miglierina, N. Ferry, and A. Solberg. Using models at runtime to support adaptable monitoring of multi-clouds applications. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*. IEEE, 2014.

[3] M. Dorigo and M. Birattari. Ant colony optimization. In *Encyclopedia of Machine Learning*. Springer, 2010.

[4] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM.

[5] O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In *Formal Aspects of Component Software*, pages 234–253. Springer, 2014.

[6] B. Porter. Runtime modularity in complex structures: A component model for fine grained runtime adaptation. In *Component-Based Software Engineering*, pages 26–32. ACM, June 2014.

[7] B. Porter, M. Grieves, R. Rodrigues Filho, and D. Leslie. RE$^{\text{X}}$: A development platform and online learning approach for runtime emergent software systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2016.

[8] B. Porter and R. Rodrigues Filho. Losing control: The case for emergent software using autonomous perception, assembly and learning. In *Proceedings of the 10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2016.

[9] T. Preisler, T. Dethlefs, and W. Renz. Structural adaptations of decentralized coordination processes in self-organizing systems. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*.

[10] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2009.