# Distributed Emergent Software: Assembling, Perceiving and Learning Systems at Scale

Barry Porter and Roberto Rodrigues Filho
School of Computing and Communications
Lancaster University
Lancaster, UK
Email: {b.f.porter,r.rodriguesfilho}@lancaster.ac.uk

***Abstract*—Emergent software systems take a reward signal, an environment signal, and a collection of possible behavioural compositions implementing the system logic in a variety of ways, to learn in real-time how best to assemble a system. This reduces the burden of complexity in systems building by making human programmers responsible only for developing potential building blocks while the system determines how best to use them in its deployment conditions – with no architectural models or training regimes. In this paper we generalise the approach to *distributed systems*, to demonstrate for the first time how a single reward signal can form the basis of complex decision making about *how* to compose the software running on each host machine, *where* to place each sub-unit of software, and *how many* instances of each sub-unit should be created. We provide an overview of the necessary system mechanics to support this concept, and discuss the key challenges in machine learning needed to realise it. We present our current implementation in both datacentre and pervasive computing environments, with experimental results for a baseline learning approach.**

## I. Introduction

The complexity of distributed systems continues to represent a major source of software engineering effort across design, optimisation and maintenance. These systems are both complex to build in the first place, composed of millions of lines of code, and are subjected to highly dynamic and often unpredictable deployment conditions. This leads to a well-worn path of *design-deploy-analyse-redesign*, in which a system is designed and deployed, then manually and painstakingly analysed to understand key performance shortfalls in its real deployment environment, following by redesign and redeployment – a cycle repeated regularly as user behaviour, system features, and available hardware continually change.

We propose a paradigm of *distributed emergent software* to effectively reduce the complexity of developing these systems. Using this approach, an engineer specifies an abstract goal (such as a set of input/output examples describing how the system should behave); a reward value for the system to optimise at runtime (such as throughput); and a set of environment measurements (such as request types, or memory usage) likely to be useful in classifying discrete contexts. Based on this information, a system is autonomously formed by searching for valid combinations of fine-grained building blocks of behaviour which match the abstract goal; the reward signal guides real-time machine learning to drive initial system assembly and real-time exploration of available behavioural compositions – including the *distribution* and *replication* of building blocks onto remote hosts. This simple approach drives complex decision-making, from the composition of behaviour on each node, to the placement of logic at a suitable host within a distributed system, and the co-location of complementary behaviours at the same or nearby hosts. As a result, we automate a large part of the design-deploy-analyse cycle to reduce the cost of developing complex distributed systems.

We present an overview of the concept; discuss its key challenges with a focus on machine learning; and present two real-world examples of emergent distributed systems in high-scale web-serving datacentres and heterogeneous pervasive systems. Our specific contributions are:

1) We present distributed emergent software, in which a system is autonomously formed from a large pool of potential fine-grained building blocks, then learns to distribute itself throughout an infrastructure to optimise a given reward signal. To minimise human involvement, our approach relies only on a generalised adaptive component model, combined with offline and online learning to locate valid and high-performing compositions.
2) We discuss the key challenges in building this kind of system, based on our varied experience over multiple domains, focusing in particular on the machine learning challenges involved in driving surprisingly complex decision-making from a very simple action-reward core.
3) We present two real-world case studies of our approach, one in high-performance datacentres to guide the composition of load balancers, web servers, databases etc. in changing workload conditions; and one in a pervasive computing environment in which a user's device can offload computation to surrounding compute resources.

Our work shows that the behavioural composition, and complex distribution decisions, of very different systems can be guided by the same simple action-reward core, representing a generalised system-building approach which reduces the complexity of distributed systems development. All of our source code is made available, along with instructions to repeat all of the experiments reported in this paper [1].

In the remainder of this paper we first survey related work in Sec. II, present our approach and its challenges in Sec III, and evaluate our implementation in Sec. IV.

## II. Related work

We consider two main strands of related work: distributed autonomous control and learning theory; and approaches to complexity management in distributed systems.

### A. Distributed Autonomous Control

Autonomous control in distributed systems has typically been considered as a way to coordinate the behaviour of well-defined agents to achieve a collective task. Our focus is on autonomously *assembling* the distributed system from a pool of a-priori unknown component building blocks, casting distributed autonomous decision making as a *continuous autonomous assembly problem* in which different behaviours can be selected and new behaviours can arrive at any time.

Jiang et al. [2] examine the use of explore/exploit reinforcement learning mechanics in quality-of-experience tuning for video streaming, in which CDN routing decisions are deferred to a learning-based process. This work is complementary, but we use reinforcement learning to make much more fundamental decisions about the design of a running system – by changing how individual sub-behaviours are implemented, where they are located in a network, and to what extent they are scaled out, requiring more complex learning coordination.

Pilgerstorfer and Pournaras consider the problem of decentralised combinatorial optimisation for energy management and bike sharing [3], taking a hierarchical approach with forward and backward error propagation to reduce a cost function in a statically-defined tree of agents. The problem domain that we explore can be seen as a generalisation of this, but one in which the topology between agents is part of the learning problem (as we learn how to distribute code) in a self-assembling system. This creates unique challenges in learning at runtime, starting from no information, how actions relate to one another both horizontally and vertically.

Golpayegani et al. report on a demand-response problem in smart energy grids, which can be viewed as a distributed optimisation problem with conflict resolution [4]. This models each agent as having a well-defined set of (mutually equivalent) actions and a communication topology, with an approach that uses globally synchronised knowledge sharing to control the flow of learning and decision making among agents. While this offers certainty over the outcome of the algorithm, and is one possible approach that we cover in our analysis of the learning design space, it is less practical as systems scale up.

The theory of distributed coordination and optimisation is studied in depth by the multi-agent systems community. Of particular relevance are Distributed Constraint Optimisation Problems [5], for problems that have formal constraints that must be mutually upheld in a distributed system, and Parallel implementations of a Monte-Carlo Tree Search [6] or Upper Confidence Bound (Tree) algorithms [7] to learn which actions offer the best reward. However, we are not aware of any specific solutions to high-scale distributed learning for the system model that we present in this paper; as such we discuss the particular challenges of learning in this context and an initial baseline solution for evaluation.

Finally, recent work by Diaconescu et al. [8] surveyed the general theme of authority for self-integrating systems and how authority might be composed, exploring how control can flow through a system to result in effective decision making. Our work is complementary to this, focusing on how a distributed system can be assembled and continually reassembled at a behavioural and topological level under the control of real-time reinforcement learning.

### B. Distributed Systems Complexity

The complexity of building distributed systems has attracted a wide range of research effort to try to simplify the development process. Typical approaches have been to develop new programming abstractions which hide the distributed interaction complexity, presenting a specialised programming paradigm to developers, or using layers of middleware to offer domain-specific interaction models.

In the former category, approaches such as Protelis [9] or TinyDB [10] present novel programming models which are sufficiently removed from the details of distribution that a runtime engine can fill in all of the complex low-level elements; these paradigms can be very powerful but also tend to be highly domain-specific.

By comparison, middleware solutions such as CORBA [11] or LIME [12] offer the ability to write code in a general-purpose language but sandbox the distributed elements of the program under a clean API – again allowing the complexities for distribution to be automated. These solutions have become a staple of modern software development, but their rigid layering increasingly causes significant challenges in end-to-end tuning of complex multi-tiered systems [13].

Moving away from layered designs, as more heterogeneous hosting resources become available – from cloud to edge computing and IoT devices – researchers are beginning to study how different parts of a distributed system may best be located in this spectrum. Recent work by Mehta and Elmroth [14] has explored this question analytically and offered recommendations of possible algorithms for decision-making. Our approach to emergent software is able to encompass the issue of placement decision-marking, and offers a real-time learned solution. We demonstrate this through one of our case studies which learns to opportunistically offload computation to nearby compute nodes when available.

At the extreme end of resource heterogeneity, researchers have started to explore the idea of disaggregated hardware [15] and how software systems might map onto this concept, with LegoOS being a prominent approach to offering a Unix-like API [16]. Our research takes a different direction to seamless distribution in which we leverage a strong component model, coupled with our concept of autonomous assembly driven by real-time machine learning, to offer total continuity between local and distributed systems. Our systems can begin as entirely local ones and then expand when they come into contact with available resources, learning how to migrate and replicate code to maximise utility for an given objective.

## III. Approach

In this section we first briefly introduce the component model on which our approach depends for autonomous system assembly and adaptation; we then describe our approach to distributed emergent systems and identify the key challenges of implementing this approach. We conclude the section by describing two case study systems we have implemented and the specific learning approach we use for evaluation.

### A. Background

Our approach assumes that we have a runtime component model which provides the self-describing code mechanics for autonomous system assembly and seamless, cheap runtime re-assembly for online learning. Components are self-describing in terms of their *provided* and *required* interfaces, where an interface is a typed collection of function prototypes, so that we can programmatically reason over which interfaces from different components can be interconnected to form a system.

The wirings between required and provided interfaces can be seamlessly adapted at runtime using *dynamic interposition*, in which in-flight function calls are temporarily held while a hot-swap takes place. This ensures that adaptations represent zero observable disruption to the running system. Lastly, as well as available function prototypes, interfaces can define transfer state which describes any state which should persist across adaptations of the component currently chosen to provide that interface via wiring decisions.

For this paper we use the Dana programming language [17] to implement our systems. Dana provides a very lightweight implementation of the above paradigm with fast adaptation (taking a few microseconds) so that runtime exploration of different behavioural compositions is very cheap – a system can be in near-constant flux without impacting the user experience. Dana also offers a fully generalised implementation of the component-oriented concept so that every element of a system – from TCP sockets up to graphical elements – can be expressed in the same paradigm. Coupled with our distributed emergent system approach, this provides a powerful general theory of fully learned system compositions and distributions.

### B. Distributed Emergent Systems

Our overall approach is illustrated in Fig. 1, in which we consider two main phases in the overall lifecycle of a Distributed Emergent System: *offline* and *online*.

In the offline phase, a functional goal is defined using a high-level abstraction (such as input/output examples, or natural language descriptions), and a search process locates all compositions of behaviour which meet this goal. This process may use both existing component building blocks, from generic library behaviours, and may also require the generation of new behaviours to fill in gaps for this specific goal. We assume that the generation of new behaviours can either be done by human engineers or by an automated synthesis process. We do not focus on the details of the offline phase in this paper, simply assuming that a set of functionally correct compositions of behaviour is made available. We assume that building blocks are fine-grained, at the level of a

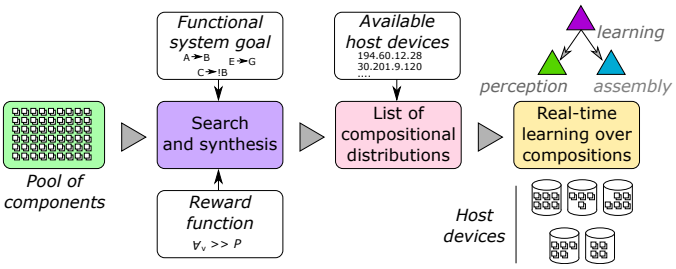


Fig. 1. Our overall approach, which uses a pool of behavioural potential implemented as components, together with a chosen reward function, to derive a set of learning actions for both local and distributed decision making.

hash-table or sorting implementation, facial feature recogniser, data format parser, etc. By injecting behavioural variations into a repository of fine-grained building blocks – such as alternative sorting algorithms – we gain a rich behavioural landscape over which to search and learn at runtime.

In the online phase, a real-time learning process searches over the set of available compositions to locate the most suitable one for each set of deployment conditions in which the system finds itself. This process is supported by an assembly module, perception module, and learning module, resident on each host, and requires three major elements: a **reward signal**, an **environment signal**, and a set of **actions**.

The reward signal is fed by live metrics from the deployed system, which report aspects of its health in real-time – such as its response time or relative quality of service. The system administrator decides how to combine multiple possible metrics into a single reward signal for learning. The environment signal is fed by a set of events from the deployed system which report aspects of the deployment environment such as the amount of energy remaining on a host, or the request pattern to which the system is currently being subjected. These events are fed into a classifier to quantise the deployment environment, which each classified environment then having a separate instance of machine learning state which learns how to maximise rewards in the respective environment. Metrics and events are both collected by our per-host perception module, which then offers aggregate data querying to a learning module.

The set of actions to accompany this runtime data is provided automatically by our assembly module which searches over possible behavioural building blocks and offers a list of valid system compositions – each with a unique identifier. Our framework first derives a set of local compositions, then a set of distributed deployment options, as follows.

For the local element, our assembly module starts from a 'main' component for the target system and extracts its required interfaces. Each such interface may have more than one available implementing component – providing the same interface but in a different way. Each implementing component is enumerated, with the required interfaces of *those* components then examined to recursively build a set of compositions which represent a diverse set of behavioural choices. Each behavioural choice (i.e., each full system composition) is represented by a unique identifier with the format:

`{comA,comB,comC|ndxA:intfType:ndxB, ...}`

This describes the entire component graph for one composition, starting with the specific list of implementation components, followed by the list of wirings between those components expressed as the indices of two components in the list and the interface type which wires them together.

We then augment the set of available local composition choices with a set of automatically-generated distributed actions. These actions are made available for *every interface* used in a system, allowing a locally-designed system to seamlessly become distributed. Our core distribution actions are *relocate* and *replicate*. To relocate a component, we generate a proxy of its provided interface with a client/server remoting pair; the proxy forwards all function calls over a network connection to a remote host, which hosts the server side of the proxy pair and forwards received function calls to the local component that is now resident at that host. To replicate a component, we generate a similar proxy but include a load-balancer (and potentially a state manager for stateful components) which decides to which of a set of possible remote copies each function call should go. Replication can also be used to implement sharding for stateful components. By using a generalised underlying component model, and keeping the granularity of our components relatively small, we can use this technique to generate a large search space of distribution actions. The complete action list for learning is thus generated by scanning for available remote hosts, and generating a relocation and replication proxy for each host so that these proxies appear as regular components implementing a given provided interface. Our composition identifiers are then expanded into the form:

$\{$`A,B`$_X\{$`hostIP`$\}$`,C|ndxA:intfType:ndxB, ...`$\}$

Where $B_X$ represents a proxy component to a remote host, the specific IP address of which is included in the composition.

A learning module on any host can examine the list of composition identifiers and select one to try; our system then calculates a difference between the currently in-use composition and the selected one, and triggers a sequence of individual adaptations (component hot-swaps, achieved by loading a new component and re-wiring a required interface to connect to it) to reach that composition. When we take an action (composition choice) which distributes part of our system to a remote host, that newly-included remote host itself gains its own local set of compositional options, rooted at the particular provided interface that has been distributed, over which to perform its own learning process as a set of actions for alternative compositions – which may include further distributing sub-elements of the local system.

The result of this is a rich decision-making process (for local compositions, and how and where to distribute code, including what to colocate) rooted in a simple flat list of actions and their observable rewards. Because all learning (and decision making) takes place in a live production environment, we learn based on actual experience of deployment conditions and how they really effect the system; this avoids issues in trying to predict deployment environments and also allows the system to respond effectively to the unexpected.

| Action | Reward |
|---|---|
| 1: {A,B,C,D\|0:iq:1,1:ir:3,...} | |
| 2: {A,Q,F,N\|0:iq:1,1:ir:3,...} | |
| 3: {A,Q,F,R,V,P\|0:iq:1,1:ir:3,...} | |
| 4: {A,Q,$F_X${192.20.50.1},P,M\|0:iq:1,1:ir:3,...} | |
| 5: {A,Q,$F_X${192.20.50.5},P,M\|0:iq:1,1:ir:3,...} | |
| 6: {A,Q,$F_X${192.20.50.5;192.31.3.1},P,M\|0:iq:1,1:ir:3,...} | |
| 7: {A,Q,X,$F_X${192.20.50.1; 192.20.50.5},P,M\|0:iq:1,1:ir:3,...} | |

Fig. 2. A learning table with composition choices as actions, including compositions with distributed elements. The reward column is populated at runtime based on experience in the deployment environment.

### C. Challenges

The set of general challenges in local emergent software systems apply equally to the distributed case, including self-referential fitness landscapes, relative rewards, and perception errors [18]. Added to this is a significant set of challenges in distributed real-time reinforcement learning, which push the limits of the state of the art solutions in this domain.

There are two core challenges in learning: how we quickly navigate a very large search space of behavioural potential (which includes local variations and distribution choices) to be able to respond quickly to new conditions that are detected; and how we coordinate learning in a distributed system to converge on a global (rather than local) objective while allowing a system to scale up to thousands of nodes.

To aid in the discussion we consider the action list shown in Fig 2. Here we have a set of local composition choices representing a system, plus a set of distributions which take particular interfaces and make them (and their sub-tree of dependencies) remote or replicated at available remote hosts. Each choice has an associated reward which is observed at runtime by the particular learning agent with this action list.

*1) Search space:* Our first challenge is that the search space for online learning may be very large, thanks to the combination of remote hosts on which to deploy each interface. In local systems, the search space can already grow combinatorially due to the permutations of component variants as a system scales up – requiring novel thinking on how learning reasons about each component [19]. When a set of additional hosts are introduced, to which we can relocate or replicate any interface, we add a second dimension of combinatorial growth.

As this search space grows larger, the number of permutations for online learning can become intractable to search over in real-time, reducing the responsiveness of the system to newly detected environment conditions. While any response speed may arguably still be faster than a human engineering team manually analysing the system and developing corrective behaviours, finding innovative solutions for the online learning problem can have a major impact in search time. These may include transfer learning [20], in which common elements of hosts, components or deployment environment elements are detected to avoid re-learning everything for an environment or host that is very similar to one we have already experienced. Much of the current research in transfer learning leaves the human operator to analyse exactly what may be transferrable

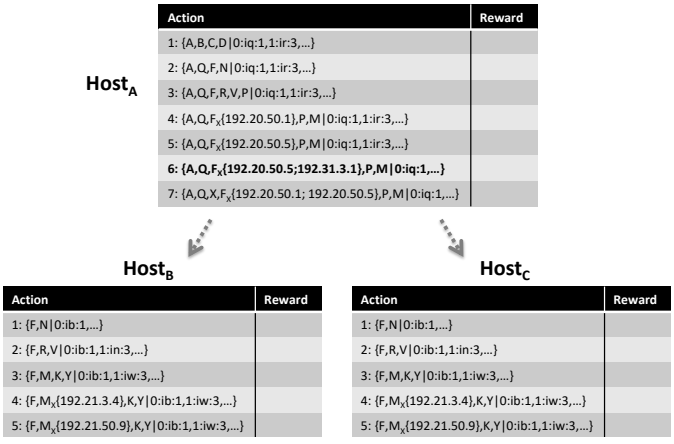


Fig. 3. Distributed learning tables with respective composition choices.

under which conditions, however, and so fully automated solutions to this problem are desirable for generality.

*2) Coordination:* Our second challenge is that coordination may be needed to ensure that the distributed system as a whole converges towards a global objective, avoiding cases in which locally-good decisions are made which have a major negative impact on the global picture when combined.

Consider the distribution of a set of learning tables shown in Fig. 3, in which $Host_A$ has selected action 6 which causes one interface (and the sub-tree of components from that interface) to be replicated across two remote hosts such that those hosts have their own set of composition choices.

Within this model we can have what we term 'vertical' and 'horizontal' interference in the learning process.

**Vertical interference** If we assume that each host has its own learning agent, and there is no coordination between them, then we can have a situation in which $Host_B$ is able to select local action 4, which provides the highest locally-observable reward, but overall provides a worse global reward than if host $Host_B$ had chosen action 2 and host $Host_A$ had chosen action 7. This phenomena is comparable to local minima from classical machine learning theory.

To assure that we converge on a global objective under these conditions we can take two different approaches: offline analysis; or global sharing of either reward and/or actions. The first and simplest is to use offline analysis to ensure that there are no local minima. Formally, if we choose a distribution action at host A, and that distribution action is globally the best thing to do for at least *one* of the remote composition choices at host B, then *every* remote composition choice must be better than any other composition choice at A. This avoids coordination messages between different learners and so has excellent scalability; however, it can only be achieved by statically removing all options at host B (using prior offline analysis) that would make the distribution worse, leaving only those that make it better. Unfortunately performing this removal by static analysis may not always be possible in a way that does not require a mock execution of the entire system in every environment, which could be prohibitively expensive.

The second approach requires adding further information sources for our learning agents: global reward knowledge, and global action knowledge. Additionally, by introducing this information, we also necessarily introduce some degree of *synchronisation* in actions taken by distributed learners.

Considering the first information source, we can add a 'global reward' column to our learning tables to better understand how local choices affect the big picture. This reward would need to be periodically disseminated to all nodes by some entity with access to the overall objective of the system. When making local choices, learning agents can then correlate their local rewards against a global reward and so understand that certain local-good actions are bad in the big picture. While workable in some cases, this approach has two potential problems: global reward might not arrive with predictable timing and so it may be difficult to correlate against a specific local action; and the globally-observed reward might appear to change erratically even when the same local action is taken. The latter scenario would indicate that there is a major source of dependency on the actions that other agents are taking.

This leads to our second source of extra information, which is adding a form of context to our learning tables to inform us of which actions *all other* learning agents took during the action that a local agent has just taken. This removes the noise problem from a global reward signal, because it explains under which exact global state a given global reward happened in relation to a local action. The only remaining source of noise is then typical natural variance in a reward signal. Again, however, a higher level of synchronisation is needed between learning agents in order to ensure that the global action context supplied to a local learning agent is accurate in the timing with which it took place relative to a local agent's action.

**Horizontal interference** The second interference source when making distributed decisions is 'horizontal'. Rather than being caused by direct dependencies between poor behavioural choices, this variant is indirect and caused by colocation of different parts of a system on the same host.

In this scenario, consider that the action lists at $Host_B$ and $Host_C$ both have an option to distribute an interface to a remote $Host_D$. If either of B or C chooses this action, the effect may locally be observed as positive; while if both choose the action at the same time, the effect may locally be perceived as negative. If B and C are not communicating about their relative choices, this situation will be perceived as inexplicable noise around this particular action and may need information sharing (either global-reward or global-action) in order to detect the explanation behind the noise; further to this, it requires a *negotiated decision* between B and C as to which of them is able to use the distribution action to D.

In all of these distributed learning challenges, the overall objective is to minimise or eliminate coordination and messaging between different learning agents in order to scale up to system sizes that are possible without a learning element. How this is achieved is an open question – and a generalised one that we believe is critical for the community to address for a large range of distributed autonomous systems.

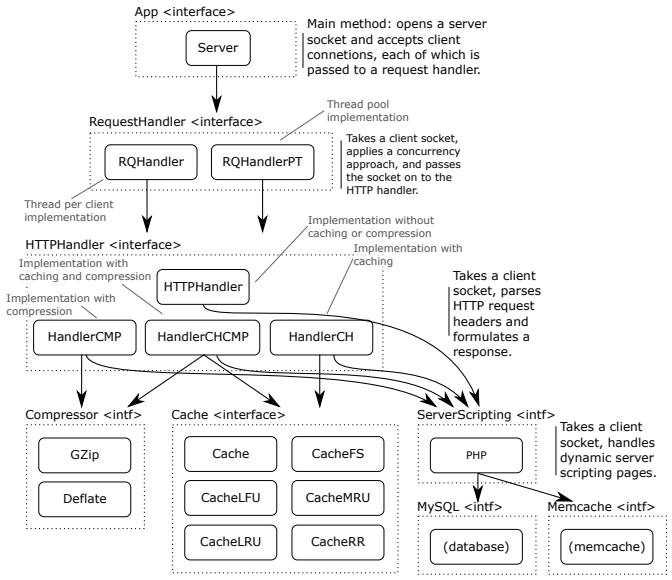


Fig. 4. The core components making up the web server element of our architecture; the memcache and database interface represent external systems that are similarly decomposed into building blocks and variations.

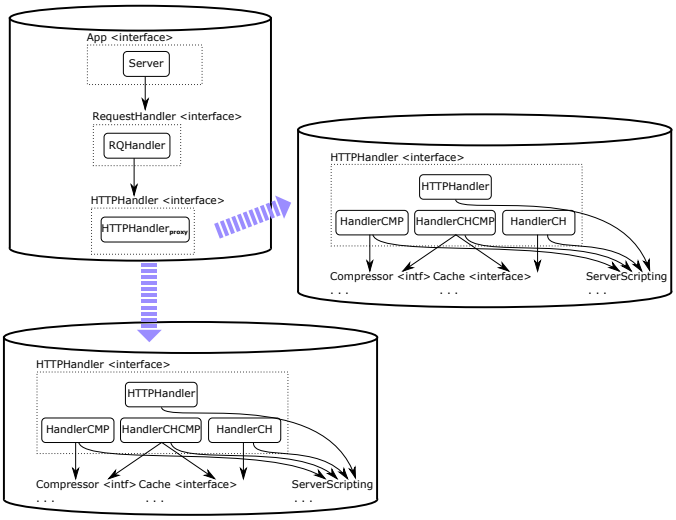


Fig. 5. A possible set of deployment decisions, reaching by selecting particular local and distribution actions from a composition list; there the `HTTPHandler` interface has been chosen for replication, and deployed (along with the rest of the sub-tree) to two additional host devices.

### *D. Case study systems*

In this section we demonstrate how two very different systems benefit from the same approach to emergent distribution and optimisation of behaviour – all driven from a single reward signal and a real-time learning approach taking simple actions that map on to complex compositional decisions.

*1) Large-scale web-serving datacentres:* The design, implementation, deployment and ongoing maintenance of the software within web-serving datacentres has become extremely complex, with a large number of interacting systems exposed to continually changing conditions. As an example of this, the backend systems supporting Facebook are such that a single page request typically hits hundreds of cooperating subsystems across a datacentre [21].

We implement an emergent version of a datacentre infrastructure using load balancing, web servers, memcache clusters, and a database engine. A subset of the component building blocks for these systems are shown in Fig. 4. As a reward signal we use average response time, and as an environment signal we use the request types that arrive at the system which are classified into request patterns.

We deploy this system in a datacentre by starting all services on a single host machine, which is the entry point of the datacentre (the public IP address to which web requests for a given site are directed). This entry-point host machine measures the overall response time of each request, and also monitors and classifies the current deployment environment.

The learning process on this server then begins to experiment with different actions, which can map either to local composition changes such as adapting the caching strategy, or to distributed composition changes such as moving or replicating a particular component across selected other servers. A typical strategy here is to replicate the web serving elements of the system (represented by the `HTTPHandler` interface) across other servers and then load-balance requests across these replicas. Relocating the database to its own host may also improve performance, as can tuning where the memcached service is deployed and how many replicas are in use. Some of these possible decision points are represented in Fig. 5.

Each decision is derived from simple runtime machine learning actions, and can be measured as good or bad in terms of a single reward signal. Whenever a new host is included into the system, a new local learning process is started on that host to learn how best to optimise the sub-composition of components given to that host, including both local decisions and further distribution choices. Because all decision making is performed in the real deployment context, based on how the system actually responds to its deployment conditions, we remove the need for manual human analysis and offline redesign, as is common practice today.

*2) Heterogeneous pervasive systems:* With increasing research into smart cities, and IoT platforms in general, alongside edge computing services and mobile devices, there is ever-more pervasive computation capability in the built environment. With this capability comes the challenge of making effective use of these devices, which in many cases can be summarised as making a tradeoff between locality (latency) and computation power (execution time) – all while the relative demands placed on each available device are in constant flux as users move and usage patterns change.

We explore one point in this design space in the context of mobile phone usage with image analysis. The image recognition capabilities of modern smartphones are increasingly advanced, with real-time translation and augmented reality assistants. We have built an emergent version of this type of system, with the basic elements shown in Fig. 6. As a reward signal here we use average processing time for image frames,

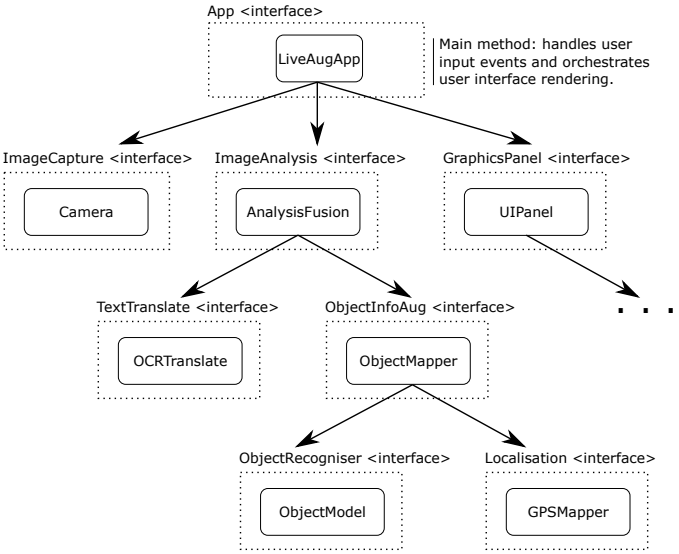


Fig. 6. The core components of a camera-based augmented reality app on a smart phone, including live text translation in a scene and annotations on objects such as buildings.

and as an environment signal we use the number of hosts available and their current free resources.

When moving through an environment, the user may come into contact with edge computing devices to which computation can be opportunistically offloaded, if there is a benefit to doing so in terms of the transmission cost of data versus the relative gain in compute speed and battery life.

This can be modelled in almost the same way as our datacentre example, using real-time learning over a set of compositional choices. In this case, however, the edge computing devices are transiently available and so the pool of host machines is more frequently updated. Even so, the problem is the same: given a set of available host machines, and the set of building blocks available from which to construct our system, can we learn which composition is best in each environment that we encounter – including which distributions of components work best for our objective.

### *E. Learning approach*

Considering the challenges discussed in Sec. III-C, in this paper we explore one of the simplest approaches to real-time machine learning in distributed emergent systems. This can be used as a benchmark for more focused studies of learning.

Our approach is based on a multi-armed bandit [22], which is the most well-studied theory on how to optimally govern the explore/exploit tradeoff in a reinforcement learning problem to maximise long-term reward. In particular we adopt a version of the Upper Confidence Bound (UCB1) algorithm, which models *reward* and *confidence* for each action, using a logarithmic model to describe confidence. The specific UCB equation that we use is shown below:

$$\sqrt{\frac{\ln n}{n_k} \cdot \frac{1}{4}\left(\frac{2\ln n}{n_k}\right)}$$

Where $n$ is the total number of actions taken, and $n_k$ is the number of times the particular action $k$ has been taken. This equation describes the confidence level of action $k$, and the result of the equation is divided by an exploration constant $c_e$; the resulting value is then added to the average reward seen for action $k$. When asked for the next action, UCB returns the action with the highest resulting value for this final addition. A learning algorithm following this equation tends to try every action once, and then begins to increasingly focus on the high-scoring actions as confidence in those choices grows.

In addition to the learning algorithm itself we also use a real-time classifier to quantise the deployment environment in which the emergent system is currently operating. Our classifier takes any event stream and derives distinct environment classes from it by using a grouped threshold approach: the set of labeled value points (such as 'memory: 150') seen in an event stream are compared against all other classified environments to see if any environment has the same set of value labels, and if so has an equivalent set of values to within a given threshold. If it does, we consider the environment to be the same, otherwise we create a new environment for this set of data points. We integrate this with UCB by instantiating a new copy of UCB, with its own learning table of actions / rewards, for each new environment detected by our classifier.

Connecting everything together, we represent every compositional choice in our emergent system as an action for UCB, and define a fixed-size observation window after which we collect the *environment class* and the *average reward* (e.g., response time) seen by the system. We use the environment class to decide which instance of UCB to select (including creating a new one for a newly detected environment). We then push our collected reward into this UCB instance for the action (composition) the system took during this time and ask the same instance of UCB which action to take next. We adapt the running system to that action/composition and then wait for the end of the next observation window.

When an action correlates with a composition which distributes an interface implementation to a remote host, we send to that host (if needed) the set of components $S_T$ which can be used to implement the interface being distributed, plus all components that are potential dependencies of $S_T$. We then start a new learning process on the remote host to learn how best to compose the delegated sub-tree of the system.

When more than one host is in use by the system, we model the distributed learning problem such that every host is given the same observation window size, and we do not share any information between different learning processes so that our coordination overhead is zero. To deal with vertical interference without coordination, as discussed in Sec. III-C, we make the simplifying assumption that local minima conditions do not exist. We enforce this by performing targeted offline testing between each pair of components to detect and remove reward conflicts during the selection of a particular distribution action.

To deal with horizontal interference we avoid 'hidden colocation' of components on the same host, where two hosts B and C can both decide to distribute one of their sub-

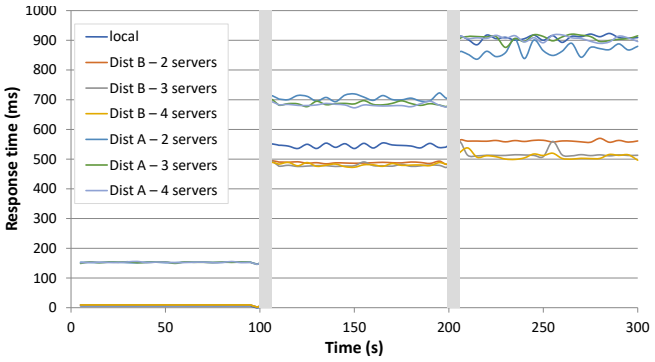


Fig. 7. Response time measured across three different environments, using increasing amounts of parallelism in the request pattern and requests that take increasingly long to serve in server-side processing scripts.

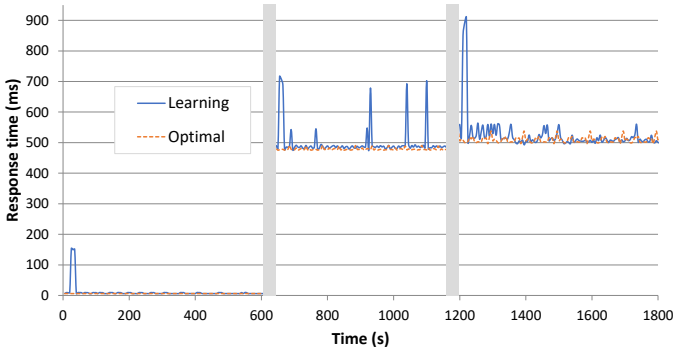


Fig. 8. Response time under real-time learning measured across three different environments, compared against the known ground-truth optimal for each case.

interfaces to a third host D. We again prevent this by using static analysis of the system composition graph to detect the particular learning actions which may provoke them and then disallow these distribution choices at runtime.

## IV. Evaluation

### A. Datacentre scenario

For this example, we deploy our emergent web serving infrastructure into a real datacentre and issue a varied series of user request workloads to observe their effects. Our rackmount servers each have Intel Xeon Quad Core 3.60 GHz CPUs and 16 GB of RAM, running Ubuntu Server 18.04. All of our servers were located in the same rack, with a shared rack head switch. Similar-spec machines were used as clients to generate workloads, with the client machines located on a different subnet (in a different building) to the servers.

We first conduct a series of experiments to locate a ground truth. Here we issue each of our workload request patterns and try each possible composition in turn, recording the average response time. This tells us which composition is really best for each workload. The results are shown in Fig. 7, which focuses on seven different compositions: everything-local; and two distribution points of the system (Dist A and Dist B) distributed to an increasing number of remote servers.

Here we see three different request patterns over time, with the average response time of each composition in each of the three phases. For the first workload (time 0 to 100), the all-local composition has the best overall response time; here all parts of the system, from the web server to the script engine and database, are located on a single server at the entry-point to the datacentre. This is best because the first workload has a dominant request proportion for static content, which does not benefit from application-level load balancing (of the type offered by our framework) since the latency of passing the content between the web server and load balancer outweighs the time taken to load the content from disk.

In the second workload, the best composition uses three servers, with the server-side script logic distributed and load balanced. This is selected because our second workload has a higher proportion of dynamic content requests, where server-side scripts are executed and communicate with a database to construct the page content. This kind of content serving is more CPU-intensive, and so the relative gain of extra CPUs on remote hosts tends to outweigh the cost of sending the data between a load balancer and replicated web server entity. The final workload shows a very close case where two compositions (using 3 or 4 servers) are almost equally good, with borderline gain from the 4th server.

We next use our learning algorithm, starting with no information, and begin to issue a set of requests to the system. The results of this are shown in Fig. 8, using the same three workloads. Here we show two data series: the known ground-truth optimal from the first experiments, and the current response time of the emergent system as it explores and learns about itself in its current environment.

The first workload, in which the everything-local composition was best, shows a very clear behaviour from the learning algorithm: we begin with a period of exploration, then reach a very high level of certainty that the local composition represents the best choice, such that no other compositions are re-explored after the initial exploration stage. In the second workload we see more varied behaviour, representing a lower degree of certainty about the ideal choice; this follows the ground truth data showing tight groups of composition choices in this workload. The initial exploration phase is therefore followed by slightly more regular re-explorations of less ideal compositions, shown as spikes in the response time – this indicates that the learning algorithm's certainty model sees a higher level of variance in the data and less distinction between available actions. Despite this, for the vast majority of the time it maintains a profile that sits on the known optimal response time. Finally, the third workload shows a noisier signal for the optimal response time itself, due to the closeness of the best two compositional choices shown in the ground truth. The learned emergent version shows a similar level of noise, but after an initial level of higher uncertainty it remains very close to the ground truth optimal throughout the experiment.

These results demonstrate that we can successfully learn how best to distribute a composition across the servers in a datacentre, depending on the current request pattern, using a

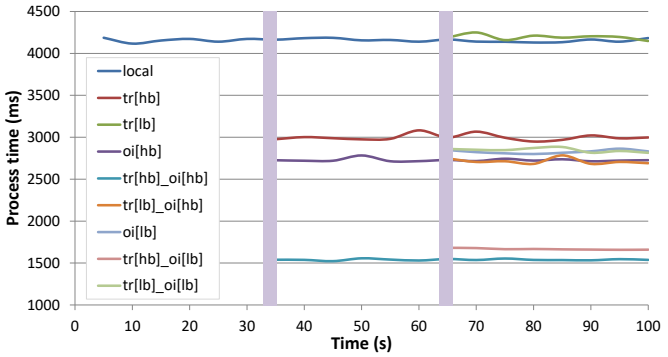


Fig. 9. Processing time measured across three different environments, with only the local host available; with a high-bandwidth in range; and with both a high-bandwidth and low-bandwidth in range.

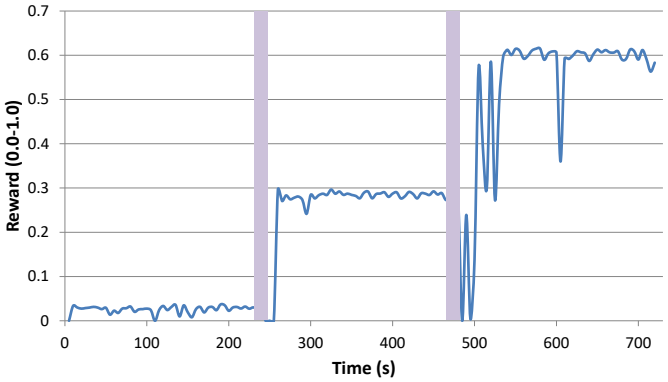


Fig. 10. Real-time learning across three different environments, showing how reward converges over time following the confidence model of UCB.

highly generalised approach to making complex distribution decisions driven by a simple reward signal. The way in which we use UCB1 means that the actual time taken to converge is linearly proportional to the number of composition choices, which will scale poorly to very high numbers of options; we leave this as an open challenge for future work.

### *B. Pervasive systems scenario*

We use this example to study the distributed emergent systems problem in a more theoretical way. We can model the basic design space here in terms of the *data volume* passed to each function in the system and the *computational intensity* of that function. In general, function calls with a low data volume and high computational intensity are worth distributing. Each interface is a collection of such functions, and the overall value of distributing an interface to a remote host is a result of the ratio of function calls on that interface. If, for example, an interface has one function $F_x$ with a high volume of data and a low computational intensity, and another function $F_y$ with a low volume of data and a high computational intensity, the ratio with which these two functions are used by the system will determine whether or not it is valuable to opportunistically offload the interface to a nearby higher-power edge device. An average ratio of 1:500 for $F_x$:$F_y$ may make this worthwhile, while a ratio of 1:1 may make it a poor choice.

In a real system, this choice is modified by the current environment in terms of the bandwidth available between the host device and an edge device (which impacts the speed and cost with which data is transferred for function calls); and by the actual portion of CPU power currently available on the local and edge device, which represent the relative gain in the computational intensity element. This environmental dependency converts the decision from one that could be modeled statically into one that needs runtime data. In these experiments we model the end-user device as a Raspberry Pi, and the edge device as one of our rackmount servers.

We again run a ground truth experiment first, in which we simulate three different environments: one with no available edge devices; one with a high-bandwidth edge device in range; and one with both a high-bandwidth and a low-bandwidth edge device in range. In these experiments the workload of the system itself is constant, with the environmental changes being the number of available devices onto which components can potentially be distributed. The results are shown in Fig. 9, which focuses on compositions in which either or both of the TextTranslate (tr) and ObjectInfoAug (oi) interface are distributed to an edge device and whether that device currently has high or low bandwidth (hb or lb) available. Here we see quite different choices in the three environments; in the first with no edge devices, the composition with all components on the end-user device is obviously the best. For the second environment, with a single high-bandwidth edge device, there are several compositional choices which represent different distributions of components onto the edge device versus the end-user device; the clear ideal choice is 'tr[hb]_oi[hb]'. In the third environment we have a larger number of composition choices with three devices available; again the ideal is 'tr[hb]_oi[hb]' but there are far more mid-way options including a distribution choice which is worse than all-local.

We next use our distributed emergent systems framework to try to learn this composition, using only data about processing time, the current environment (defined as devices in range with their CPU potential and available bandwidth), and the set of compositions available (which is this case transiently includes distributions of components to edge devices when available). The results of online learning are shown in Fig. 10, again divided into the same three environment cases explored in the ground truth experiments

In this graph we show the reward level, rather than the processing time, to give more insight into the internal view of the learning algorithm. This reward is gained by first normalising the reported processing time into a value between 0.0 and 1.0, then inverting the processing time (which is really a 'cost') into a reward. A higher value here is therefore better, and in each phase the highest reward level corresponds to the optimal choice seen in Fig. 9.

For the first phase we see the environment in which only the user device exists, where the reward signal shows natural variation as computation speed fluctuates. In the second environment we spend a short period of time exploring poor

options and so have a very low reward level, then quickly reach a high level of certainty about the best action and so see a major and sustained increase in reward. In the third environment we see a similar initial exploration period, followed by a slower rise in the reward level and occasional reward drops. The latter two effects are caused by the higher number of close compositional choices. Again, however, the overall reward level is high and sustained for the majority of the experiment, despite natural variations in the reward signal.

## V. Conclusion

We have presented a novel approach to constructing a distributed system in an emergent way, in which both the choices about local behavioural choices on each host, and on which hosts to deploy each available sub-compositional graph, are unified under a simple action-reward model.

This paves the way towards a significant reduction in the complexity of distributed systems design and deployment by allowing a machine learning algorithm to orchestrate, in a live deployment setting, the location, replication factor, and per-host design, of an entire distributed system.

We have also presented the main challenges in achieving this, focused on the difficulty of distributed learning. In particular, the twin problems of locating a global optimal solution while minimising message passing and coordination for high-scale systems represents a very challenging design space – and one which is highly relevant to a large class of autonomous systems, yet is relatively under-explored.

We have implemented our approach in two real-world demonstration systems, covering a datacentre web-serving infrastructure and a pervasive systems environment. These examples demonstrate the potential generality of the concept in everyday settings that experience continuous environmental changes, and so benefit from continuous learning of the assembly of behaviours used to compose a system. Our implementation uses one point in the space of possible distributed learning solutions, as a baseline against which to understand potentially more advanced future approaches, which shows the initial costs of constructing confidence models followed by good convergence in both example systems.

## Acknowledgements

This work was partly supported by the UK Leverhulme Trust via the *Self-Aware Datacentre* project, grant RPG-2017-166.

## References

[1] Source code from this paper with instructions: http://research.projectdana.com/saso2019porter.

[2] J. Jiang, S. Sun, V. Sekar, and H. Zhang, "Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 393–406.

[3] P. Pilgerstorfer and E. Pournaras, "Self-adaptive learning in decentralized combinatorial optimization - a design paradigm for sharing economies," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2017, pp. 54–64.

[4] F. Golpayegani, I. Dusparic, A. Taylor, and S. Clarke, "Multi-agent collaboration for conflict management in residential demand response," *Computer Communications*, vol. 96, pp. 63 – 72, 2016.

[5] F. Fioretto, E. Pontelli, and W. Yeoh, "Distributed constraint optimization problems and applications: A survey," *J. Artif. Int. Res.*, vol. 61, no. 1, pp. 623–698, Jan. 2018.

[6] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," in *Computers and Games*, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 60–71.

[7] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293.

[8] A. Diaconescu, B. Porter, R. Rodrigues Filho, and E. Pournaras, "Hierarchical self-awareness and authority for scalable self-integrating systems," in *International Workshop on Self-Improving System Integration*. IEEE, September 2018, pp. 1–8.

[9] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. New York, NY, USA: ACM, 2015, pp. 1846–1853. [Online]. Available: http://doi.acm.org/10.1145/2695664.2695913

[10] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: An acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, Mar. 2005.

[11] T. Bennani, L. Blain, L. Courtes, J.-C. Fabre, M.-O. Killijian, E. Marsden, and F. Taiani, "Implementing simple replication protocols using corba portable interceptors and java serialization," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 549–.

[12] A. L. Murphy, G. P. Picco, and G. . Roman, "Lime: a middleware for physical and logical mobility," in *Proceedings 21st International Conference on Distributed Computing Systems*, April 2001, pp. 524–533.

[13] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 217–231.

[14] A. Mehta and E. Elmroth, "Distributed cost-optimized placement for latency-critical applications in heterogeneous environments," in *2018 IEEE International Conference on Autonomic Computing (ICAC)*, Sep. 2018, pp. 121–130.

[15] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 249–264.

[16] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 69–87.

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

[18] R. Rodrigues Filho and B. Porter, "Defining emergent software using continuous self-assembly, perception, and learning," *Transactions on Autonomous and Adaptive Systems*, vol. 12, no. 3, pp. 1–25, September 2017.

[19] B. Porter, M. Grieves, R. Rodrigues Filho, and D. Leslie, "Rex: A development platform and online learning approach for runtime emergent software systems," in *Symposium on Operating Systems Design and Implementation*. USENIX, November 2016, pp. 333–348.

[20] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, Oct 2010.

[21] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song, "Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services." in *OSDI*. USENIX, 2016, pp. 635–651.

[22] S. Bubeck and N. Cesa-Bianchi, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Foundations and Trends in Machine Learning*, vol. 5, no. 1, pp. 1–122, 2012. [Online]. Available: http://dx.doi.org/10.1561/2200000024