Autonomous State-Management Support in Distributed Self-adaptive Systems

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

***Abstract*—Modern systems are increasingly required to be adaptable in order to handle constantly changing operating environments. Adaptability, in the majority of work in this area, is based on the ability to adapt the behaviour of a running system where multiple implementations are available. Example of this are technologies such as reflective middleware and meta-models which offer introspection and control over how logic is wired together. While these technologies support high degrees of autonomous flexibility around the *compute* element of distributed systems, they completely neglect handling *state* in an externally-managed, automated way. This paper advocates a reflective model over system *state*, to complement existing models that enable meta-management of behaviour. This concept has the potential to support an entirely new dimension of self-adaptive systems, offering a richer set of options to compose a system. We demonstrate a possible implementation of this concept by extending a lightweight component-based model; our implementation can transparently and generically relocate, replicate, and shard stateful components. Using a set of annotations, our framework constructs a pool of possible compositions which distribute any system using a variety of different state management options; at runtime, we can dynamically learn which state distribution approach best suits each environment. We posit that this offers a new and unexplored dimension of self-adaptive systems, supporting novel concepts such as self-distributing systems which can emerge to best match their environment.**

***Index Terms*—reflection, meta-model, state management, self-adaptation**

I. Introduction

Modern systems are increasingly required to handle the high levels of volatility present in their operating environments. This has motivated a wide range of research in self-adaptive and autonomous systems. However, the majority of work in this area is based on the ability to adapt the behaviour of a running system where multiple implementations are available; for compositional adaptation this is based on technologies such as reflective middleware [10] and meta-models [16] which offer introspection and control over how logic is wired together. In *distributed systems*, these ideas are partially operationalised in technologies such as cloud and edge computing which use a variety of container technologies to support flexible infrastructures which can scale up and down as needed and place containerised behaviour at different locations.

While these technologies support high degrees of autonomous flexibility around the *compute* element of distributed systems, they lack support for handling *state* in an externally-managed, automated way. Developing systems with microservices, for example, forces engineers to manually separate state from components, using a particular database technology. By doing this, stateless components are then free to be autonomously relocated and replicated across an infrastructure as needed. This state separation task remains highly manual to engineer, requires careful design of distributed state consistency models that best match the goals of the system, and in deployment leaves systems with only one fixed way to handle their state – making it hard to explore alternatives that could increase performance across different deployment conditions.

In this paper we advocate reflective models over system state, offering new degrees of flexibility in distributed system design and new dimensions of self-adaptive systems research as a result. Successfully implementing this would enable engineers to express state and behaviour naturally together in the same components, enabling autonomous frameworks to automatically and simultaneously manage both the distribution of behaviour *and* state as coupled system elements. For example, by knowing which functions in an interface cause state updates and which only read state, we can begin to understand how to automatically replicate a component – including its state – and which distributed state consistency models are viable. Likewise, by modeling how interfaces manage collections, we can automatically shard stateful components across hosts, using generic proxies to find which state belongs to which shard for each request. This paper has two contributions:

1) We identify system state as a forgotten element of runtime reflection, and motivate why this is a crucial research direction for future self-adaptive systems;
2) We present one possible reflective meta-model for state which offers a set of new degrees of flexibility for autonomously reasoning about and (re)composing distributed system with stateful components;

The remainder of this paper is structured as follows: Sec. II describes related work. Sec. III describes our extension to a component-based model, showing how a simple reflective state model can aid in autonomous state management via relocation, replication and sharding of stateful components. Sec. IV concludes the paper.

II. Related Work

Our work draws inspiration from the reflective middleware [10] community which applies reflection as a basic operation to enable adaptation on software systems at runtime. Although this was widely used to support systems adaptation in different domains (e.g. mobile applications [6] and QoS [4]), there is no work to our knowledge that applies reflection to enable autonomous management of components' *state* at runtime to specifically support a range of possible distributed systems compositions. This goal is particularly useful in modern systems where cloud [12] and edge computing [15] technologies, which make infrastructure more flexible, are widely adopted.

In contrast to our approach, prevailing technologies go in the opposite direction and try to eliminate state from most components. Stateless components can be freely replicated or placed on any machine across the infrastructure, while state is completely and manually separated into specialized components such as databases. This approach is most clearly embodied in the recent Microservices trend [7], which are combined with technologies such as Mesos [9], Kubernetes [2], or serverless computing [11], as the 'de facto' standard in creating large-scale web-based systems for handling stateless compute elements which can take full advantages of the flexible infrastructures supported by the cloud.

Underpinning these frameworks is the technology of containers [1]. This is analogous to a 'component model', which isolates self-contained deployable software modules, enabling automated replication and placement of parts or the entire system across a wide range of compositions to deal with a wide range of real-time challenges. For scalability, containers are used to encapsulate microservices, for example, and to replicate them as incoming workload increases. For network latency, containers are used to ship self-contained software to run on the network edge. While this presents very useful degrees of freedom for automated management of *behaviour*, none of these technologies offer automated *state* management.

We propose a major step-change to this thinking, in which a meta-model automatically and transparently supports externalised state management in a *unified* way with the computation (or behaviour) of a distributed system. We present this model as an extension of the common characteristics of runtime component models, offering a new form of reflection for system state at runtime, and show how this meta-model can support a range of existing scaling and consistency models over system state according to the current context.

III. Approach

Self-adaptive systems are key to handle the increasing levels of volatility in contemporary operating environments. For single-host (local) systems that use compositional adaptation, where different pieces of logic are swapped in and out to change the system's behaviour, runtime component models are a popular implementation choice. These include frameworks written in an existing programming language such as Fractal [5], OpenCom [3], and OSGI [8], and full programming languages specially designed to support continuous adaptation such as Dana [13]. These component-based approaches support runtime adaptation of fine-grained components, enabling seamless runtime architectural adaptation of software systems.

We use runtime component models as a foundation for our state meta-model because they already support dynamic system composition in a well-formed way – though they do not tend to consider issues of distributed systems or state management. We extent runtime component models to support distribution of their components transparently, so that any component can be distributed across a network without having to be explicitly programmed for this purpose. Rather than using container technologies or microservices which have no explicit dependency or service description model, this approach allows us to leverage runtime reflection over strongly-typed interfaces in order to automatically analyse the set of possible compositions of a given system and ensure soundness of that system as we adapt between different compositions.

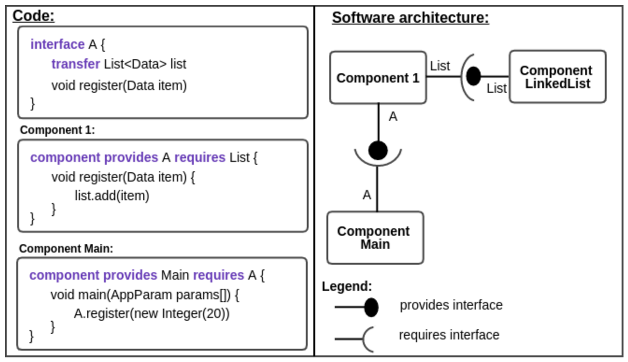
By extending runtime component models to support transparent migration of components between hosts, we also open the possibility of transparently replicating components for load-sharing in scale-out environments, and of sharding components to more dynamically load-share among popular state regions. To achieve these capabilities, however, we require more than a meta-model to describe the abstract logic of components via their interfaces; we also require a meta-model to abstractly describe the *state* stored in each component so that we can reason over how to distribute, replicate, or shard this state in a safe way. This allows any local system, which tend to be naturally programmed in a way that tightly integrates state with logic, to benefit from fully automated distribution and a wide range of possible distributed state management approaches depending on the current context.

The realisation of this vision has the potential to support fully automated decision-making about distributed systems design, including placement and replication of behaviour *and* management of associated state in approaches such as traditional databases, NoSQL-style solutions, distributed object stores, or cache clusters, depending on the constraints around each element of state and the current deployment context.

In this section we describe our prototype model for transparent state management in distributed systems, using the above concepts. We first define our assumptions on the features that a local/single-host component model should support to enable our approach; for this paper we chose the Dana component-oriented programming language to illustrate key features of the model. We then define a distributed model, which is an extension of the local model and supports replication and relocation of stateless components, before defining a new meta-model that supports autonomous reasoning about state which enables transparent distribution of stateful components.

A. Local Model

We assume that a local component model consists of a set of rules used to develop lightweight component-based software. These rules are interpreted by a framework or a language runtime to support composition of software out of small



Code:

```
interface A {
    transfer List<Data> list
    void register(Data item)
}
```

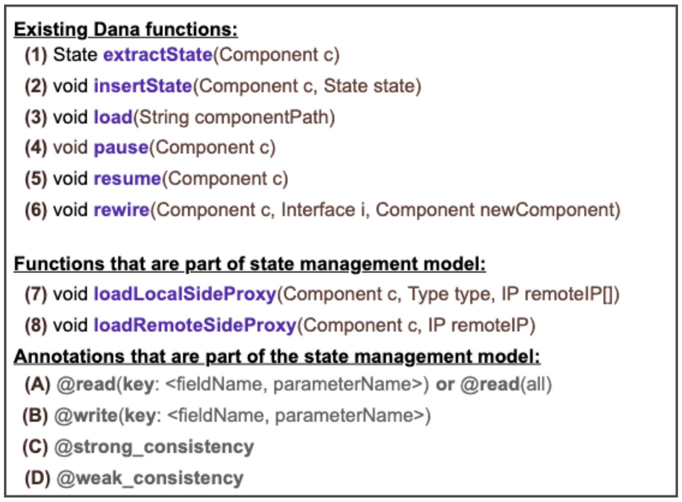
Component 1:

```
component provides A requires List {
    void register(Data item) {
        list.add(item)
    }
}
```

Component Main:

```
component provides Main requires A {
    void main(AppParam params[]) {
        A.register(new Integer(20))
    }
}
```

Fig. 1. Component and interface's anatomy. On the left side, there is an example of an interface and two components. On the right side, there is an illustration of three connected components to form an executing system.



Existing Dana functions:

(1) State **extractState**(Component c)
(2) void **insertState**(Component c, State state)
(3) void **load**(String componentPath)
(4) void **pause**(Component c)
(5) void **resume**(Component c)
(6) void **rewire**(Component c, Interface i, Component newComponent)

Functions that are part of state management model:

(7) void **loadLocalSideProxy**(Component c, Type type, IP remoteIP[])
(8) void **loadRemoteSideProxy**(Component c, IP remoteIP)

Annotations that are part of the state management model:

(A) @**read**(**key**: <fieldName, parameterName>) **or** @**read**(all)
(B) @**write**(**key**: <fieldName, parameterName>)
(C) @**strong_consistency**
(D) @**weak_consistency**

Fig. 2. A list of Dana runtime functions and annotations that are used to transparently support state management.

components and the hot-swapping of components at runtime without breaking the system. This could be implemented in a component model such as Fractal, OSGi, OpenCom, or Dana.

In these, each component must provide (i.e., implement) a given interface, and may require a collection of interfaces (i.e., dependencies) used to assist their implementation. Given a collection of components, with their required and provided interfaces, a runtime can connect compatible components together to form a fully functioning system. Fig. 1 illustrates this basic anatomy of components and composition.

Besides the highlighted keywords **interface**, **component**, **requires** and **provides** that define interfaces, the component implementation, and which interface is provided and required by a component respectively, we also assume the presence of a concept for which Dana uses the syntax **transfer**. The **transfer** keyword defines the state that every component which implements a given interface requires, such as the label displayed on a GUI button, or the set of items stored in a hash table. This exposes all such state to the component runtime, and during the adaptation process this state can be extracted from the old component and injected into the replacement component, preserving key state across adaptations.

The adaptation process itself is orchestrated through a set of special operators, shown in Fig. 2, and supported by the component runtime through architectural reflection. We briefly illustrate the local-system adaptation process using the components and interface shown in Fig. 1, which links to how we implement transparent distribution of components in the following section. Consider the procedure to replace $Component_1$ with $Component_2$ (a new component that is not illustrated in Fig. 1) while the system is executing. The adaptation process starts by loading the new component into memory (using function (3) in Fig. 2). We then pauses the execution of $Component_{Main}$ (function (4)) while we extract the state from $Component_1$ and insert it into $Component_2$ (functions (1) and (2) respectively). At the end of the state transferring action, $Component_2$ is connected to $Component_{Main}$ (function (6)). $Component_1$ is then unloaded from memory and finally $Component_{Main}$ is resumed (through the execution of function (5)). This adaptation process supports replacement of stateful components and is crucial in supporting the development of seamless adaptive software [14].

The ability to adapt behaviour in useful ways comes from the fact that interfaces define an abstract version of a piece of logic which can be implemented in multiple different concrete ways – such as an interface which defines sorting operations and different implementing components using different sorting algorithms, or an interface which defines a GUI button widget and different implementing components offering different graphical button styles. In order to make state handling generic across adaptations, we rely on the transfer syntax to similarly define abstract state of an interface in such a way that the implementation is free to use its own internal data structures to represent that state. A hash map interface, for example, may define `put` and `get` operations and an array of key/value pairs as the state of the component; an implementing component may then choose to represent this key/value pair list in a flat array, using a multi-level index, or some other internal data structure. Whenever we adapt away from a component, we therefore expect that component to translate its internal data structures into the abstract state defined in the interface, and the reverse for a component being adapted into.

B. Distributed Model

Our distributed model extends the local one to transparently support two new dimensions of adaptation in any system: the *relocation* of a component to a remote host, offloading its computation demands, and the *replication* of a component across multiple remote hosts to scale out and load balance. These two additions enable systems to be entirely written as a local system, and during execution have their local components moved to other host machines to form a distributed system. In this section we only consider these actions for the simple case of stateless components (i.e., those that declare no `transfer` state), covering the stateful case in the following section.

Component relocation is the action of transferring a locally-executing component to another available host machine. Replication, on the other hand, is the action of transferring multiple copies of a locally-executing component to a set of distinct machines. Both share two main stages that the system needs to execute: i) replace the target local component to a local *proxy* component, and ii) load the target component (or multiple replicas of it) in the remote machine(s).

The first stage is executed by replacing the component to be relocated with a proxy. This process uses exactly the same adaptation process supported by the local model, where two local components are exchanged. The difference is that the newly added component is a proxy component rather than an implementation of the interface. Proxy components act as regular components that implements the same interface as the component they replaces, except that for each function they transparently perform remote procedure calls (RPC) to the actual component, now running remotely. In the case of replication, the proxy component also implements a load balancing strategy (e.g., round robin) which forwards each incoming function call to a different replica.

The second stage is to load the moved (relocated or replicated) component onto the new machine(s). This stage consists of loading a remote-side proxy component (function (8) in Fig. 2), loading the relocated component and satisfying all its dependencies (i.e. to connect it to all components it requires to properly function) on the remote machine. The remote-side proxy is a generic component which acts as a server to receive the local-side proxy calls and forwards them to the right function on the remote component. For replication, this remote loading process is simply followed for each replica.

Once these two adaptation stages are completed the system is resumed and all calls for the relocated/replicated components are performed as if they were local function calls. Note that the transparent distribution mechanisms described here require that any failures caused by remote interactions must also be dealt transparently, since components are not expected to be designed with errors cases in mind for remote interactions. The implementation of failure handling is beyond our scope here but a range of different approaches is possible.

C. State Management

In this section we discuss an extension to our distributed model (Sec. III-B) to support relocation and replication of *stateful* components (i.e., those that have `transfer` state declared), plus sharding of state among remote component replicas where possible.

A key challenge in supporting these actions is to appropriately handle *consistency* and inter-dependencies in state across replicas or shards. A major element of this challenge is to support *reduced consistency* wherever possible in order to enhance scalability and flexibility of autonomous state management. As an example of why this is useful we can consider a set of replicated copies of a stateful component, for which we can easily inject the same copy of state into each replica when it is first started. For all following function calls, we can then adopt a strong consistency model and simply replicate the function call on each replica, ensuring their state transitions are always synchronised. However, this approach clearly does not enhance the throughput or performance of a system as each replica is involved in servicing every request. The ability to adopt a weaker consistency model where possible (i.e., where the design of a component and its relationship with state allows) is therefore crucial to gain more flexibility for different higher-performance distributed state management solutions.

In this section we cover each style of distribution, and present a set of keywords that programmers can use to annotate parts of their interfaces to support greater flexibility in autonomous state management. The set of annotations that we use for this purpose are shown at the bottom of Fig. 2.

1) Relocation: As previously described, this is the action of moving a local component to another machine in the infrastructure. The actions needed to relocate a stateless component comprises of i) replacing the target component to a proxy, and ii) loading the target component remotely with all its dependencies. At the end of this process the target component and all components that were connected to it are unloaded from the local machine and are loaded and properly connected in the remote machine. Depending on the component, a good portion of the local component graph gets relocated to a new machine. For stateful components, relocation also has to consider transferring state from the old instance of the local target component to its remote instances. Considering an entire subgraph of components might get relocated, all stateful components that are part of this subgraph also require their state to be transferred to their new remote instances.

State transfer always occurs after the new remote instances are loaded and before resuming the local component. Once transferring is complete, the paused component is resumed and the system continues to operate as it was when all components were local. The relocation action only moves one instance of the component and its subgraph to a different machine, and therefore, just as in the local case, all stateful components only exist in one place and no further attention are needed to maintain consistency. All relocation stages, including state transferring, is done by the runtime and it only requires component developers to define transferable state with the **transfer** keyword. During state transferring, the runtime extracts all state, and after the new remote instances are created, it injects them in the new instances.

2) Replication: This is the action of relocating multiple instances of one local component across the infrastructure. In other words, the set of actions used to perform relocation is basis to perform replication. For the stateful component, the result of this action can also mean the replication of state. However, to allow for more flexible system's compositions (i.e. different ways in which to support **horizontal scale**), our model, in this case, also support the extraction and centralisation of component's state so that the component behaviour can be replicated numerous times without causing any state inconsistency. The two cases are detail below.

We name 'full replication' the action of replicating both component behaviour and its state multiple times across the system's infrastructure. In full replication, the stage of transferring state from the local component to its remote replicas are conducted the same way as in the relocation operation, but repeated multiple times for all existing component's replicas.

The proxy that replaces the local component instance can be implemented in two different ways, which give more flexibility to the system's composition. One option is to have a proxy that forwards incoming function calls to all replicas, implementing the concept of group communication, increasing reliability and availability of systems. Another option is to implement a load balancing strategy that forwards each individual incoming function call to a replica.

In case of the latter proxy implementation, the system has also to transparently provide a consistency model that would propagate the changes that happen in the state of one replica to all replicas. Note that depending on the consistency level required, for each operation processed by a replica the changes in the state has to be propagated to the other replicas, somehow maintaining the order of changes, which may result in large overhead. Less constraining consistency models (e.g. eventual consistency), which guarantees that the system will be consistent in a point in the future, but not after each operation, can be applied, but only when components have high tolerance to stale data. The definition of the allowed consistency model is defined by the component's developer when creating the component itself, and adding annotations (c) or (d) (in Fig.2) in the interface on top of the state definition.

Finally, extracting and isolating state allows for free replication of components without having to deal with consistency models or group communication. This process is done by relocating the component that encapsulates the state before replicating the target component. At the end of the relocation stage, the target component that is still running locally already has the proxy component that remotely interacts with the component holding its state. When the replication action occurs, it proceeds just as if the component to be replicated were stateless. All replicas end up with the same version of the proxy that interacts with the state. The local proxy component that forwards incoming function calls uses a round robin load balancing strategy to forward calls to the replicas without worrying about state consistency.

3) Sharding: This action is very common in databases and consists of splitting a data set among distributed replicas. This action allows stateful components to be replicated and each of the replicas hold one part of the component state. This is particularly useful when distinctive and independent part of the state are intensively required. Then by splitting them and relocating them, we can reduce competition to computing resources, for instance, CPU in case access to this disjoint part of state is split, or network when access to this part of the state can be split and placed closer to the system element which is constantly requesting it. Therefore, when forwarding function calls to component replicas, the proxy has to know which part of the state the called function will be using, in order to properly invoke the function on the correct replica. It is important to notice that sharding can only be performed in component with a collection of items as state. For instance, lists, queues, hashtables, and so on, but are not restricted to these basic data types, any component holding a collection of items as their state is a candidate for sharding. There are three main stages that need to be executed to realise state sharding, they are: data splitting, state transferring, and load balancing strategy definition.

Data splitting is the process of separating the local aggregated set of items to be transferred, each sub set of the collection, to a set of remote machines. The general mechanism to perform this action is to use a consistency hashing approach. This approach uses a key, for each item in the collection, and the number of replicas to distributed the items over. Then we iterate over the items applying a hashing function to the key which will result in the number of the replica where the data will be inserted. The entire state splitting process is performed locally, once the items are split in different states, they are then transferred to the remote machines. The main problem in this stage is that the component's developer has to provide a field that will be used as the item's key, using annotations (a) or (b) shown in Fig. 2, on top of every function.

After the first stage is concluded, the split state is then transferred to their corresponding replicas. The state transferring stage is performed as in other cases (e.g. replication and relocation). Each shard of the state is transferred to a specific replica following the consistency hashing approach.

The third and final stage aims to guarantee that the calls are being properly routed to the right replica. This requires the developer to provide the key of the data that will be accessed or changed as a result of the function execution. Once developer annotates all functions, the proxy can use the provided key and the same consistency hashing scheme to determine which replica should be invoked to execute the remote call.

As a final note, all previous state management actions here described can be used recursively. That is, when a set of component are replicated along with their state (in case of 'full replication' previously described) the same set of state management actions can be further carried out by, for instance, sharding the state of the newly created replicas and so forth. This makes the model extremely flexible, supporting a large set of possible system compositions that are useful when supporting requirements for a variety of operating conditions.

This recursive use of state management actions is possible because replication, relocation and sharding are realised by a set of primitive actions and properties ensured by the annotations and primitive functions (illustrated by Fig. 2), which are always available to newly created replicas.

D. Performance Analysis: Horizontal Scaling

Our described approach supports a wide range of ways in which to compose a system – allowing any local system, including stateful components, to be converted into a distributed one which can automatically benefit from horizontal scaling. In this section we present the main enabled composition styles and discuss the scenarios in which they may boost a system's

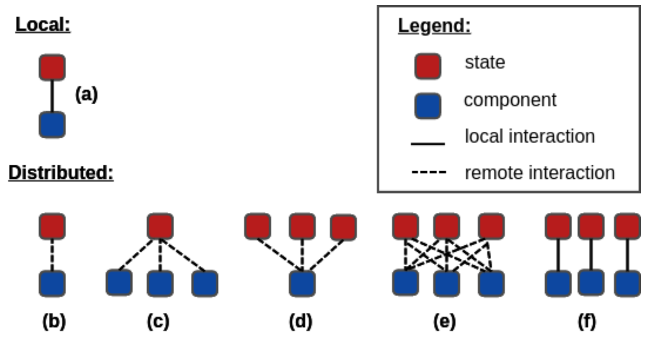


Fig. 3. Distinct compositions of stateful component relocation, replication and sharding enabled by the model.

performance by automatically supporting horizontal scaling. Fig.3 illustrates the different compositions.

The top left of Fig.3 shows two components connected locally. The lower component has the upper component as its state, and the interaction between them is performed locally. The bottom half of the image illustrates all possibilities of relocation (b, c), replication (c, d, e, f) and sharding (d, e, f) for this pair of components. We will use these compositions as illustrating examples of how changing the interaction between components and their state can impact system performance.

The system compositions illustrated in (b) and (c) consist of extracting and isolating state. This is a widely used system architecture that separates behaviour from state, and enables behaviour to be freely replicated (as shown in (c)), decreasing system's response time when the amount of processing time spent on each replica is high, due to parallelism in handling incoming requests. Although this is an architecture that enables replication of the behaviour part of the component, it centralises access to state, making state access a bottleneck. The impact of this bottleneck is noticeable on the system's performance when time-consuming update operations are constantly made by all replicas.

A better composition is to use the same system arrangement but to change the proxy: instead of forwarding function calls to all replicas, it shards the state and redirects requests to only the replicas that host that part of the data. Sharding state, in general, can boost performance if different parts of that state are intensively updated, and the update processing time is high. Sharding state in this situation improves performance by splitting the state management workload among replicas.

Another system composition supported by state sharding is (e), which has the same effect as (d), but enables replication of the component behaviour. This is useful in cases where updating state is time-consuming on the component that holds state *and* the processing time on each of the component behaviour is also high.

These examples demonstrate some of the wealth of system composition permutations which can automatically be gained using our approach, taking any purely-local system and allowing it to be distributed or horizontally scaled by managing its stateful components using our reflective state meta-model.

IV. Conclusion

This paper presents a reflection over state management in modern systems. We argued that popular modern technologies that are used to develop systems that handle highly dynamic operating environments lack a holistic support for state management. Instead, engineers are forced to consider state in a case-by-case basis, making it difficult to support adaptation when changes to operating environment requires different ways in handling system's state. We propose a reflective model over system state that offers new degrees of flexibility in distributed system design, and new dimensions of self-adaptive systems research as result. We hope the community join us in exploring a complete version of this model, and to tackle the challenges imposed by implementing such model.

Acknowledgements

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

References

[1] C. Anderson. Docker [software engineering]. *IEEE Software*, 32, 2015.
[2] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.
[3] Gordon Blair, Geoff Coulson, Jo Ueyama, Kevin Lee, and Ackbar Joolia. Opencom v2: A component model for building systems software. *IASTED software engineering and applications*, 2004.
[4] Gordon S. Blair, Anders Andersen, Lynne Blair, Geoff Coulson, and David Sánchez. Supporting dynamic qos management functions in a reflective middleware platform. *IEE Proceedings-Software*, 147(1):13–21, 2000.
[5] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.
[6] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on software engineering*, 29(10):929–945, 2003.
[7] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017.
[8] Richard S Hall, Karl Pauls, Sturat McCulloch, and David Savage. Osgi in action. *Creating Modular Applications in Java*, 2011.
[9] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.
[10] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, June 2002.
[11] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.
[12] Peter Mell et al. The nist definition of cloud computing. 2011.
[13] Barry Porter. Runtime modularity in complex structures: A component model for fine grained runtime adaptation. In *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*, pages 29–34, 2014.
[14] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. Rex: A development platform and online learning approach for runtime emergent software systems. In *Symposium on Operating Systems Design and Implementation*, pages 333–348. USENIX, November 2016.
[15] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.
[16] Jian Wang, Keqing He, Bing Li, Wei Liu, and Rong Peng. Meta-models of domain modeling framework for networked software. In *Sixth International Conference on Grid and Cooperative Computing (GCC 2007)*, pages 878–886. IEEE, 2007.