Self-Managed Decentralised Systems using K-Components and Collaborative Reinforcement Learning

Jim Dowling
Distributed Systems Group
Trinity College Dublin
jim.dowling@cs.tcd.ie

Vinny Cahill
Distributed Systems Group
Trinity College Dublin
vinny.cahill@cs.tcd.ie

ABSTRACT

Components in a decentralised system are faced with uncertainty as how to best adapt to a changing environment to maintain or optimise system performance. How can individual components learn to adapt to recover from faults in an uncertain environment? How can a decentralised system coordinate the adaptive behaviour of its components to realise system optimisation goals given problems establishing consensus in dynamic environments? This paper introduces a self-adaptive component model, called K-Components, that enables individual components adapt to a changing environment and a decentralised coordination model, called collaborative reinforcement learning, that enables groups of components to learn to collectively adapt their behaviour to establish and maintain system-wide properties in a changing environment.

Keywords

Decentralised Self-Adaptive Systems, Collaborative Reinforcement Learning, Architectural Reflection

1. INTRODUCTION

The specification of system-wide properties is a good starting point for the construction of self-managing distributed systems, as the system can use them to reason about system behaviour and can adapt itself to actively establish and maintain them, e.g., using self-management actions. Minsky describes system-wide properties as regularities in a system [1] and examples include fault tolerance and load-balanced. System-wide properties may be formal properties of the system determined at design time or attributes of the system that are established and maintained at run-time. Existing design time techniques that can introduce system-wide properties into distributed systems do so in a top-down manner, decomposing system behaviour and making it amenable to formal analysis [2]. These include constraints in software architectures and formal models such as π-calculus. At run-time systems typically rely on centralised or consensus-based approaches to establish and maintain system-wide properties, using techniques such as group communication protocols [3] or centralised configuration managers in dynamic software architectures [4]. Both centralised and consensus-based techniques require communication overhead to establish agreement on the value of shared variables, and as a consequence of network dynamism, the physical limits of network latency and the possibility of partial failure they are not viable for decentralised environments [5].

Recently, there has been a trend towards using decentralised control techniques to build self-adaptive, decentralised systems, with areas such as peer-to-peer [6] and ad-hoc networks [7] producing noticeable achievements. A common pattern for decentralised distributed system architectures is to model them as a collection of frequently similar, coordinating agents where each agent gathers information on its own, takes independent decisions on how to behave and communication between agents is localised [8] or through some shared environment [9]. The benefits of such an approach include improved robustness and scalability, the lack of centralised points of failure, the potential for system-wide self-management as well as possible evolution of the system through evolving the local rules of the agents [8].

The construction of decentralised distributed systems with system-wide self-management properties presents a number of challenges. These include developing a component model that can support local self-management behaviours, designing decentralised coordination models for components that can establish and maintain system-wide properties over collections of components and determining the functionality required at the component-level to build such decentralised coordination models.

2. THE K-COMPONENT MODEL

Many self-management behaviours can be engineered using self-adaptive components, as components have the ability to change their behaviour or structure at run-time in order to accomplish specified goals [10], e.g., adapt to discovered faults or sub-optimal performance. Self-management behaviour for components requires the active monitoring of component states and external dependencies for events that cause adaptation actions, such as reconfiguring connections to faulty components.

The K-Component model is a framework for building self-adaptive systems [10] that operate in a decentralised environment. The model provides a component interface definition language called K-IDL, an extension to IDL, that supports the definition of component states and adaptation

WOSS '04 Oct 31-Nov 1, 2004 Newport Beach, CA, USA

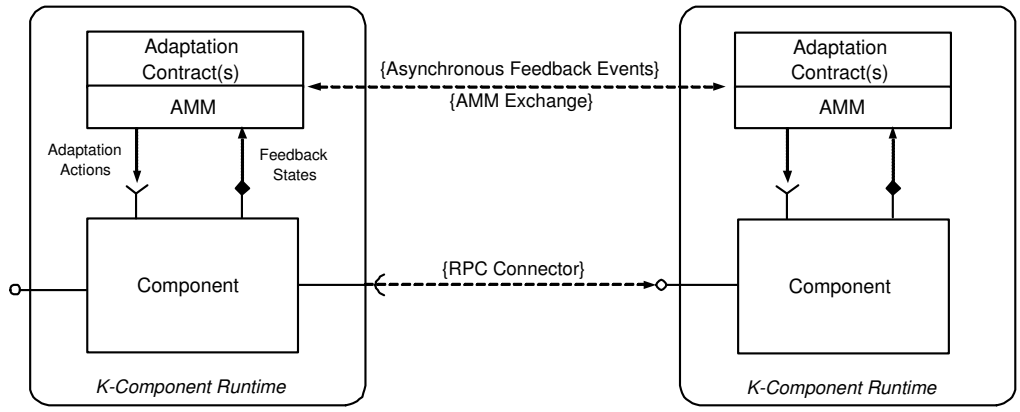

Figure 1: Connected Components in different K-Component Runtimes

actions, as well as required interfaces. Component states and adaptation actions are used by decision making programs to reason about component operation and adapt component operation. A K-IDL compiler translates component interface definitions into an extended version of IDL-2 and interfaces are compiled using modified Orbacus/C++ middleware.

A K-Component is a runtime with a single address space where components, defined in K-IDL, can be deployed and dependencies between components, whether internal or external to the runtime, are managed using connectors. Architectural reflection is used to reify the structure of component and connectors deployed in a K-Component runtime as an architecture meta model (AMM).

As the K-Component model is designed to enable the construction of self-adaptive software in decentralised systems, there is no explicit representation of the system-wide AMM. The system-wide software architecture is partitioned amongst the K-Component runtimes in the system. Each K-Component runtime manages its local software architecture as an AMM that describes its partial view of the system. This partial view is limited to the internal components and connectors deployed in K-Component runtime and the external components connected to components in the runtime, (see figure 1).

Reflective autonomous programs, called adaptation contracts, are associated with a component and operate on a runtime's AMM by reasoning about adaptation conditions using component/connector states and feedback events regarding remote component states. Programmers can specify adaptation contracts using a declarative programming language called the Adaptation Contract Description Language (ACDL). The ACDL allows programmers to declaratively associate component/connector states or feedback events with adaptation actions using if-then rules or the event-condition-action (ECA) paradigm. Events are used to communicate feedback information between connected components in different runtimes and are as a mechanism for building decentralised coordination models, such as CRL.

One problem with both rule-based and the ECA approach to specifying self-adaptive behaviour is that it becomes infeasible as the space of possible feedback events and adaptation actions increases. For complex, instrumented distributed systems, programmers cannot be expected to know about and handle all possible internal component and external environmental states or be able to accurately predict the outcome of executing some adaptation action in a dynamic environment. For this reason, in K-Components self-adaptive behaviour can also be learnt by components using an unsupervised technique called collaborative reinforcement learning (CRL). CRL also enables the decentralised coordination of groups of connected components for the purpose of establishing system-wide properties. The next section introduces system-wide properties for distributed systems and shows how system-wide properties of a distributed system can emerge from the interaction of its components.

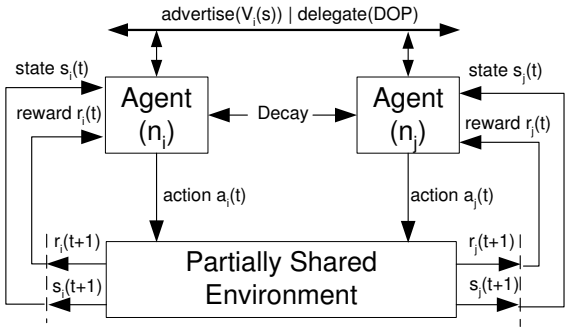

Figure 2: CRL Model

3. SELF-ADAPTIVE DECENTRALISED SYSTEMS USING COLLABORATIVE REINFORCEMENT LEARNING

CRL is a decentralised approach to establishing and maintaining system-wide properties in distributed systems. CRL is an extension to Reinforcement Learning [11] (RL) for decentralised multi-agent systems. CRL does not make use of system-wide knowledge and individual agents only know about and interact with their neighbouring agents.

In RL, an agent associates actions with system states in a trial-and-error manner and the outcome of an action is observed as a reinforcement that, in turn, causes an update to the agent's action-value policy using a reinforcement learning strategy [11]. The goal of reinforcement learning is to maximise the total reinforcements (rewards) an agent receives over a time horizon by selecting optimal actions.

Agents may take actions that give a poor payoff in the short-term in the anticipation of higher payoff in the longer term. In general, actions may be any decisions that an agent wants to learn how to make, while states can be anything that may be useful in making those decisions. As action selection is probabilistic, there is some trial-and-error in the selection of actions and RL is not a suitable technique for learning self-* behaviour for the classes of distributed system that are intolerant to suboptimal action selection, such as real-time systems.

CRL is based on a variant of the coordination model technique found in swarm intelligence algorithms where agents learn from the successes of their neighbours. It has been previously applied to optimise routing in ad-hoc networks [12]. CRL solves system optimisation problems by specifying how individual agents solve discrete optimisation problems (DOP) using RL, advertise their results to their neighbours and transfer control to neighbours by initiating the start of a new DOP on a neighbouring agent, see Figure 2. Each agent stores a cache of with DOP solution information advertised by its neighbours that represents the agent's local view of the system i.e., its neighbourhood.

In a system of homogeneous agents that solve related problems, the cached information can be used by agents to help them improve their solution to their local DOP. This process is known as collaborative feedback and it enables agents to share more optimal policies [11], increasing the probability of neighbouring agents taking the same or related actions. This process can produce positive feedback in action selection probability for a group of agents. Positive feedback is a mechanism that reinforces changes in system structure or behaviour in the same direction as the initial change and can cause the emergence of system behaviour or structure [13, 9]. In CRL, the positive feedback process continues until negative feedback, produced either by constraints in the system or the decay model, causes agent behaviour to adapt so that agents in the system converge on stable policies. The decay model provides negative feedback by degrading DOP solution information in an agent's cache over time according to a decay rate. In CRL, system-wide properties can emerge from the interaction of positive and negative feedback in the solution of discrete optimisation problems [12].

3.1 Specifying CRL Problems in the K-Component Model

In CRL, system optimisation problems are decomposed into a set of DOPs, the solution of which are performed by collaborating agents. In K-Components, adaptation contracts represent the agents and components/connectors represent the environment. Each component explicitly specifies both adaptation actions and states (indicating DOP solution information) in its interface, whereas connectors have a fixed set of states and adaptation actions. Adaptation action implementations also calculate and return a reinforcement giving evaluative feedback on the action's success to adaptation contracts. The actions[1], states and reinforcements are used by the adaptation contract (the CRL agent) to learn the optimal policy for how to adapt components given current component and connector states. Every K-Component maintains its local view of the system state by caching remote components states in its AMM. The advertisement function used in CRL to update the cache can be specified using the feedback event model in the ACDL.

[1]Architectural adaptation actions are also available to adaptation contracts

3.2 Load Balancing using CRL

In this section we introduce a simplified version of a system optimisation problem in a decentralised system: how to balance load amongst peers in the network using CRL. Our goal here is not to fully specify a solution to this complex problem, but rather to elucidate the basic behaviour of the CRL model and show how autonomic behaviour for the system can emerge from self-adaptive (local decision making) components.

A decentralised system contains a varying number of components in different K-Component runtimes and can be modelled as a partially connected graph, with components as vertices and connections between neighbouring components as connectors. A load balancing application for such a system requires, at a minimum, a description of the load and a cost metric, the load cost, that characterises the ability of a component to handle a particular load [14]. When a component receives a load, the adaptation contract for the component makes a load balancing decision for the load. It can base its decision on monitored load costs of the component's neighbours, i.e., its connected components, and use a load balancing function to execute actions to balance the load among them.

In this example, we simplify the problem by assuming all components reside in runtimes on devices that have equal capabilities and every load consumes an equal amount of resources. A load can be any particular type of resource (e.g. storage, unit of computation, etc). Every component calculates its *internal load cost,* i.e., its estimated cost to handle a particular load type, with a lower load cost indicating a better ability to handle a load. Each load has start and termination states - a component where the load is generated and a component where the load is handled. The goal of every adaptation contract in the system is to balance loads to the components with the lowest estimated load cost.

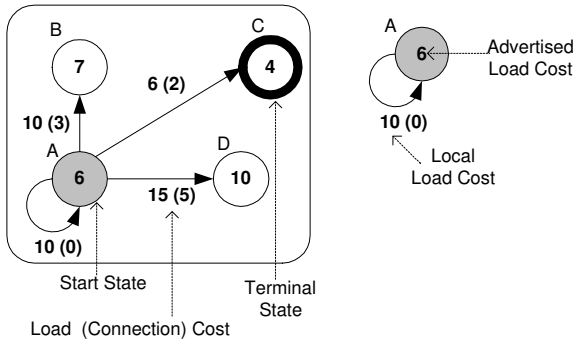

Figure 3: Load Balancing Decisions in CRL

Figure 3 illustrates the problem of how a series of loads generated by COMPONENT A are balanced over COMPONENTS A,B,C,D. A load balancing function on every adaptation contract uses a cost function to calculate its internal load cost and the ability of its neighbours to handle the load. Each neighbour has an *advertised load cost*, defined as a component state in K-IDL, but adaptation contracts also have to consider the connection cost in forwarding the load to the

neighbour. A simple linear model of a neighbour's estimated cost would be: estimated load cost = (advertised load cost + connection cost) . Components can advertise a load cost that is lower than their internal load cost if they have a neighbour with a load+connection cost that is lower than their internal load cost, see legend in figure 3.

In figure 3, the optimisation problem starts at COMPONENT A. COMPONENT A attempts to solve a discrete optimisation problem, where the cost for COMPONENT A to handle the load locally is 10, but the cost to forward the load to COMPONENT C is 6. The solution to COMPONENT A's discrete optimisation problem is to forward the load to COMPONENT C. When the load is forwarded by an adaptation contract executing the component's adaptation action, this triggers the start of a new DOP at the adaptation contract for COMPONENT C, that in turn calculates its own optimal solution, which is to handle the load locally. In this way, the system-wide load-balancing behaviour is solved as a series of DOPs.

In K-Components, a component can advertise a change in its load value using feedback events. The caching of a neighbour's advertised load costs in the AMM reduces the amount of control traffic generated in the system. Cached load costs are used by the adaptation contract's cost function, but these are only estimations of a neighbour's current loads. As such adaptation contracts make load balancing decisions based on estimations of the value of loads, rather than their true, in fact unknowable, values. This looser form of consensus introduces some problems relating to the use of stale data in decision making, e.g. loads may be balanced to the wrong component due to stale cache information, but CRL can help improve the accuracy of cached estimates through both advertisement and decay of cache information.

System-Wide Self-Management in Decentralised Systems

The proposed CRL solution presented can enable a load-balancing decentralised system to self-manage its load-balancing behaviour in the presence of node failure or unavailability. When a component fails to advertise its estimated load cost to its neighbours, e.g. due to node or communication failure, the neighbours decay their local cached load costs for that component, reducing the probability that the component will be selected as a target for load balancing. When a component becomes unavailable, the probability of it receiving load balancing requests drops proportionally with the cache's decay rate. This self-management property is not explicitly programmed in the system and is a consequence of the negative feedback on a component's cache. The load balancing problem can also be characterised at the component level as self-management decision making in an uncertain environment.

There is no attempt to achieve consensus on the load cost of different components in the system before a load balancing decision is taken by an adptation contract. Each component takes load balancing decisions based on the estimated loads of other components. In the case where a group of components converge in their estimation of the load of a particular component, we can say that the nodes achieve emergent consensus on the load of that component. Advertisement of load costs helps reduce uncertainty in load-balancing decision making by making more information available to the adaptation contract about the state of the system. As a result, the advertisement of estimated load costs helps adaptation contracts improve the learning of useful self-management behaviour.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we describe approaches to engineering self-management properties in distributed systems at the component level with K-Components and at the system level using CRL in K-Components. We believe that many self-management properties can be engineered at the component level using a self-adaptive component model and that self-management properties at the system level can be modelled and solved as system optimisation problems, such as CRL. Collaborative learning of a self-managing decision policy by components can provide a system with flexibility and robustness enough to establish system-wide self-management properties in both predicted and unforeseen environmental conditions.

5. REFERENCES

[1] N. H. Minsky, "On conditions for self-healing in distributed software systems," *Proceedngs of the Autonomic Computing Workshop, AMS '03*, 2003.

[2] A. Montresor, H. Meling, and O. Babaoglu, "Towards self-organizing, self-repairing and resilient distributed systems," *Future Directions in Distributed Computing*, vol. LNCS 2584, 2003.

[3] M. Hayden, "The ensemble system," *PhD Thesis, Cornell University: Dept. of Computer Science*, 1997.

[4] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the first workshop on Self-healing systems*, pp. 27–32, ACM Press, 2002.

[5] R. Khare and R. N. Taylor, "Extending the representational state transfer (rest) architectural style for decentralized systems," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2004.

[6] I. Clarke, S. Miller, T. Hong, O. Sandberg, and B. Wiley, "Protecting free expression online with freenet," *IEEE Internet Computing, Jan/Feb*, 2002.

[7] E. Curran and J. Dowling, "Sample: An on-demand probabilistic routing protocol for ad-hoc networks," *Technical Report: Department of Computer Science, Trinity College Dublin*, 2004.

[8] J. Kennedy and R. Eberhart, *Swarm Intelligence*. San Francisco, California: Morgan Kaufmann, 2001.

[9] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: from natural to artificial systems*. New York: Oxford University Press, 1999.

[10] J. Dowling and V. Cahill, "The k-component architecture meta-model for self-adaptive software," *Proceedings of Reflection 2001*, 2001.

[11] R. Sutton and A. Barto, *Reinforcement Learning*. MIT Press, 1998.

[12] J. Dowling, E. Curran, R. Cunningham, and V. Cahill., "Collaborative reinforcement learning of autonomic behaviour," *2nd International Workshop on Self-Adaptive and Autonomic Computing Systems*, 2004.

[13] S. Camazine, J. Deneubourg, N. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau, *Self-Organization in Biological Systems*. Princeton University Press, 2003.

[14] M. Jelasity, A. Montresor, and O. Babaoglu, "A modular paradigm for building self-organizing peer-to-peer applications," *Proceedings of ESOP03: International Workshop on Engineering Self-Organising Applications*, 2003.