The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems

Jim Dowling

A thesis submitted to the University of Dublin, Trinity College

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

October 2004

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Jim Dowling

Dated: 30th March 2005

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Jim Dowling

Dated: 30th March 2005

For Tony

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Vinny Cahill for his guidance, support, diligence and mentoring on the long journey. He's always been patient and prodding when I needed it. The other main acknowledgement must go to the present and past people at DSG. What a fantastic place it has been to learn and grow. Of the whole crew, I'd like to give a special mention to Raymond Cunningham, Tim Walsh, Donal Lafferty, Andronikos Nedos, Johan Andersson, Mads Haahr, Marco Kilijan, Kulpreet Singh, Vinny Reynolds, Elisa Baniassad, Simon Dobson, Stephen Farrell, Anthony Harrington, Siobhan Clarke, Christian Jensen, Rita McGuinness, Peter Barron, Greg Biegel, Stefan Weber, Rene Meier, Dominik Dahlem, Ivana Dusparic, David McKitterick, Andy Edmonds, Liz Gray and John Keeney.

From the last of the reflection team, I'd like acknowledge the originals: Barry Redmond, Tilman Schaefer and Peter Haraszti. To all those friends who endured with me, particularly during the writing phase: Linda who never complained, Brendan who never let me forget, Don, Phil, Anthony, Laura, Steve, John, Fiach, Mikey and everyone else who has offered encouragement and support.

And finally, to my family who have never once complained that I'm taking too long. To my father for his inspiration, to my mother for her unconditional love and support, to my brother Rob for his ambition, to Maeve and Aoife for keeping me grounded and sane, and lastly to Tony who will be always in our hearts and minds.

Jim Dowling
University of Dublin, Trinity College
October 2004

Summary

Distributed computing systems are moving towards increasingly autonomous operation and management, in which their interacting components can organise, regulate, repair and optimise themselves without human intervention. The emerging field of autonomic distributed computing addresses the challenge of how to design and build distributed computing systems that can manage, heal and optimise themselves given high-level objectives. Adaptive software provides some of the functionality required for building autonomic computing systems, as it allows system behaviour or structure to be changed at run-time to fulfil the high-level objectives. Self-adaptive software is a subclass of adaptive software that autonomously executes adaptation logic, code concerned with monitoring for adaptation conditions and triggering adaptation actions. This thesis proposes that self-adaptive components are a useful building block for autonomic computing systems, as they can autonomously adapt their structure and behaviour at run-time to fulfil specified goals. It also shows how decentralised coordination of self-adaptive components can establish autonomic properties for distributed systems in dynamic and uncertain environments, such as wireless ad-hoc networks or peer-to-peer systems.

Self-adaptive software requires programming support for the specification of its adaptation logic in order to avoid tangling adaptation-specific code with functional code. Reflective techniques can help modularise adaptation logic, but existing self-adaptive systems based on reflection only support the specification of adaptation logic that executes synchronously with program execution, even though events triggering adaptive behaviour are often temporally orthogonal to program execution. Also, although it is known that self-adaptive software can evolve and learn its adaptive behaviour over time through the use of information relating to past adaptive behaviour, none of the existing models have the ability to learn improved adaptive behaviour online. Finally, the use of decentralised coordination models to build distributed systems with autonomic properties from self-adaptive components has not been addressed by existing systems. Current reflective programming models for building adaptive software lack support for the separate specification of application-level adaptation logic that can learn and optimise a component's adaptive behaviour.

The K-Component model is a component framework for building self-adaptive distributed systems that addresses the aforementioned problems. Adaptation logic for components is specified in a declarative programming language and encapsulated at run-time as a set of reflective programs that are scheduled asynchronously to program execution. The reflective programs operate on an architecture meta-model and reason about adaptation conditions using events that provide feedback regarding the state of components and connectors. Adaptation logic can be specified using if-then rules or the event-condition-action paradigm and the unsupervised learning of adaptive behaviour is also supported using

reinforcement learning. Collaborative reinforcement learning is introduced as a decentralised coordination model that can coordinate the adaptive behaviour of groups of connected components for the purpose of establishing system-wide autonomic properties in dynamic and uncertain environments. A further contribution of this thesis is an asynchronous model of reflection for adaptive software that decouples the execution of reflective code from base-level code.

This work reviews existing models of self-adaptive software from the areas of reflective systems, dynamic software architecture and autonomic computing. It describes the programming model for K-Components, its architecture meta-model, a contract description language, a model of asynchronous reflection and collaborative reinforcement learning. As an evaluation of the model, a load balancing application demonstrates how autonomic distributed systems properties can emerge from the decentralised coordination of self-adaptive components using collaborative reinforcement learning. The K-Component model has been implemented as an extension to CORBA in C++.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> "The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man"
>
> George Bernard Shaw, Maxims for Revolutionists: Reason, Man and Superman (1903)

This thesis presents self-adaptive components as a building block for autonomic systems and shows how their decentralised coordination can establish and maintain autonomic properties in distributed systems that operate in dynamic and uncertain environments. Collaborative reinforcement learning is introduced as a decentralised, self-organising, coordination model that enables properties such as self-healing and self-optimisation to be established in decentralised distributed systems composed of self-adaptive components. This chapter motivates the work, introduces the self-adaptive component model and collaborative reinforcement learning, presents its contributions and finally outlines a roadmap for the thesis.

1.1 Motivation

As networked computing becomes pervasive in everyday life in the developed world, with easy access to PCs, smartphones and wireless networks, seamless networked services require more complex and larger scale distributed infrastructures. Due to the increased cost and complexity of managing such infrastructures manually, distributed computing systems are moving towards more autonomous operation and management.

Autonomic computing addresses the area of autonomous self-managing software systems. It offers a vision of future distributed systems as seas of interacting, autonomous components that organise, regulate and optimise themselves without human intervention. However, with increasing system size and complexity the ability to build autonomously managed distributed systems using existing programming languages, design techniques and management infrastructures is limited (van Renesse et al., 2003; De Wolf and Holvoet, 2003; Montresor et al., 2003; Ardaiz et al., 2003; Andrzejak et al., 2003). Top-down design techniques such as problem decomposition and modularisation result in distributed system architectures that become unwieldy and impractical with increasing system size, as they require too much global knowledge (van Renesse et al., 2003; De Wolf and Holvoet, 2003; Montresor

et al., 2003; Ardaiz et al., 2003; Andrzejak et al., 2003). Distributed applications that operate in dynamic and uncertain environments with no support for global knowledge, such as peer-to-peer and mobile ad-hoc networks, require new mechanisms to enable them to reason about their own behaviour and autonomously manage themselves without recourse to traditional top-down or centralised techniques. The motivation behind this work is to provide support for building distributed systems with self-managing properties in such decentralised environments using bottom-up techniques.

The construction of self-managing distributed systems presents a number of challenges. Firstly, a self-managing distributed system requires the design and integration of self-management functionality at many different levels, including the component, middleware and software architecture levels. Secondly, a decentralised system requires coordination between its components to enable them to organise themselves into a self-managing system.

The specification of *system-wide properties* is a good starting point for the construction of self-managed distributed systems, as the system can use them to reason about system behaviour and can strive to actively establish and maintain them using self-management actions. Minsky describes system-wide properties as regularities in a system (Minsky, 2003) and examples include deadlock-freedom, 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 (Montresor et al., 2003). These include constraints in software architectures (Allen, 1997; Oreizy et al., 1999) and formal models such as π-calculus (Milner, 1999). Systems based on these methodologies typically rely on centralised or consensus-based approaches to establish and maintain system-wide properties, and are implemented using techniques such as group communication protocols (Ellis and Gibbs, 1989; Hayden, 1997) or centralised configuration managers in dynamic software architectures (Garlan and Schmerl, 2002; Moreira et al., 2001). Both centralised and consensus-based techniques require a large amount of communication overhead to establish agreement on the value of shared variables (van Renesse et al., 2003), and as a consequence of both the physical limits of network latency (Khare and Taylor, 2004) and network dynamism (Montresor et al., 2003) they are not viable for decentralised environments.

We see autonomic properties of a distributed system as system-wide properties that contribute to its self-management. In particular, we define an autonomic property of a distributed system as:

> "An autonomic property of a distributed system is a property that contributes to its autonomous management or desired operation and is actively maintained by the system itself at run-time"

The goal of *autonomic computing systems* is to establish, verify and maintain such autonomic properties in dynamic environments with minimal external intervention. Commonly cited autonomic properties in the literature include self-healing, self-optimisation, self-protection and self-configuration (Ganek and Corbi, 2003; Kephart and Chess, 2003), often known collectively as self-star properties (Strunk and Ganger, 2003). The field of autonomic computing draws its inspiration from complexity science (Prigogine and Stengers, 1984; Waldrop, 1992; Holland, 1996; Wolfram, 2002), and in particu-

lar the human body's autonomic nervous system. Throughout this thesis, the definition of autonomic computing is taken from Kephart (Kephart and Chess, 2003) as follows:

> "[Autonomic computing systems are] computing systems that can manage themselves given high-level objectives[1]"

Autonomic computing is a new area of research, and while there are few reference architectures for autonomic systems it is envisioned that autonomic computing systems will be decentralised systems composed of interacting *autonomic elements* (Kephart and Chess, 2003; Ganek and Corbi, 2003). Autonomic elements are defined by Kephart in (Kephart and Chess, 2003) as entities that:

> "manage their internal behavior and their relationships with other autonomic elements in accordance with policies that humans or other elements have established."

In distributed systems, an autonomic element can vary from a process to an agent to a component. As this thesis deals with distributed systems composed of components, the discussion is constrained to *autonomic components* instead of the more general autonomic elements.

Kephart also states that autonomic components monitor their execution context and environment, plan actions that should be taken and execute self-management actions where necessary. Ganek makes similar claims about autonomic components, in that they "continuously monitor [component behavior] through 'sensors' and make adjustments through 'effectors'" (Ganek and Corbi, 2003). Many of these features are found in existing reflective middleware systems (Blair et al., 2001; Capra et al., 2003), reflective component models (Moreira et al., 2001; David and Ledoux, 2003) and auto-adaptive systems (Atighetchi, 2003; Kon et al., 2001), although these existing systems only provide self-management functionality at specific layers in a distributed system such as the application, middleware or architecture level.

At present there are no programming models with support for building autonomic components or decentralised systems with system-wide autonomic properties. The specification and implementation of autonomic components presents the challenges of how to specify their functional behaviour, how to specify their self-managing behaviour and how to integrate the component model with a distributed programming model. This work introduces a component model, called K-Components, for building components with autonomic properties. K-Components is based on the Common Object Request Broker Architecture (CORBA) (Henning and Vinoski, 1999) standard.

This thesis also addresses the problem of how to self-organise components in a decentralised system in order to establish autonomic properties for the system. It investigates the relationship between the autonomic behaviour provided at the component-level and the set of autonomic properties that can emerge at the system-level. In particular, it introduces a model for coordinating the adaptive behaviour of components for the purpose of establishing system-wide autonomic properties. The model is a decentralised optimisation technique called *collaborative reinforcement learning* (CRL) that can be used to build coordination models that and can establish and maintain certain autonomic properties in decentralised systems composed of K-Components.

[1] High level objectives are generally supplied by humans and can be measured and verified using some metric, e.g., balance load over a group of servers or maintain a service level agreement.

1.2 Self Adaptive Components

Adaptive software techniques enable the construction of components with many of the features of autonomic components identified by Kephart and Ganek, including the important ability to adapt system behaviour or structure at run-time in order to accomplish specified goals. Self-adaptive software (Oreizy et al., 1999; Dowling and Cahill, 2001b) is a subclass of adaptive software that contains its own adaptation logic (Dowling and Cahill, 2001b), code concerned with a system's self-adaptive behaviour. Self-adaptive components can dynamically reconfigure their structure and behaviour in response to identified states such as faults or sub-optimal operation and, as such, represent a natural model for building self-managing components.

The dynamic reconfiguration of a component-based distributed system is difficult due to dependencies that exist between components as well as dependencies between components and their execution environment (Kon et al., 1999; Moazami-Goudarzi, 1999; Wermelinger, 2000; Blair et al., 2000; Georgiadis, 2002; Almeida, 2001; Whisnant et al., 2003; Sadjadi and McKinley, 2004). A software pattern commonly found in existing self-adaptive and auto-adaptive[2] systems is the separation of reconfiguration-management functionality from system functionality (Atighetchi, 2003; Garlan and Schmerl, 2002; Ranganathan and Campbell, 2003; Dowling and Cahill, 2001b). Existing systems provide mechanisms for monitoring system states, or a representation of them, for specified conditions and executing adaptation actions that maintain adaptation consistency (Moazami-Goudarzi, 1999; Blair et al., 2001), i.e., they do not affect the integrity of the running system. Adaptation actions can vary from changing implementation strategies, planning ahead, and changing current beliefs about the state of the system to reconfiguring the system structure.

Techniques such as reflection (Smith, 1984; Maes, 1987) and dynamic software architectures (Allen et al., 1998; Kramer and Magee, 1998; Moazami-Goudarzi, 1999; Oreizy et al., 1999) provide principled means for building self-adaptive software as they can provide an open implementation (Kiczales et al., 1997) and guarantee adaptation consistency (Moazami-Goudarzi, 1999; Wermelinger, 2000; Almeida, 2001) respectively. Architectural reflection (Cazzola et al., 2000; Moreira et al., 2001; Cuesta et al., 2002b) represents a powerful synthesis of these techniques for building adaptive and self-adaptive distributed systems. This work adapts previous models of architectural reflection to support distributed systems in decentralised environments.

1.2.1 Architectural Reflection

Dynamic software architectures are an effective technique for building reconfigurable software (Oreizy et al., 1999; Moazami-Goudarzi, 1999; Garlan and Schmerl, 2002; Dashofy et al., 2002; Moreira et al., 2001; Wermelinger, 2000), as they provide models that preserve system consistency during reconfiguration. A system is said to maintain adaptation consistency if it satisfies some structural integrity requirements, if the components in the system are in mutually consistent states, and any application state invariants hold (Moazami-Goudarzi, 1999). The K-Component model enables a system's structure to be represented as a type of dynamic software architecture, an architecture meta-model.

[2] The term self-adaptive is favoured over auto-adaptive as it is a commonly used term in the area of complex adaptive systems.

The model provides components as a building block for constructing applications as compositions of components. Components encapsulates their implementation behind a strongly-typed interface and connectors to abstract and instrument a component's external interactions with other components. Components can be instrumented for adaptation by a subsystem by allowing programmers to specify feedback states, feedback events and adaptation actions on components.

In contrast to existing software architecture models of distributed systems that employ a system-wide architecture (Luckham, 1996; Medvidovic et al., 2000; Allen, 1997; Magee et al., 1995), the software architecture of a distributed system built using K-Components is decentralised (Khare, 2003), i.e., distributed among nodes in the system. Each K-Component in the decentralised system maintains a local software architecture and a partial view of the system that covers directly connected components on neighbouring nodes.

Reflection provides a principled way of constructing self-adaptive systems (Dowling and Cahill, 2001b) and separating reconfiguration functionality from component functionality (Blair et al., 2000). A system that supports *architectural reflection* reifies its software architecture, i.e., its representation as a configuration graph of components and connectors, as an *architecture meta-model* (Blair et al., 2001). The architecture meta-model is a causally connected representation of the system's software architecture and reconfiguration operations performed on the architecture meta-model are reflected in the system's base-level, i.e., the actual components and connectors of the system. The architecture meta-model is not only concerned with the architectural features it reifies but also with an associated set of architectural constraints, describing how and when to safely reconfigure the software architecture (Blair et al., 2001). Reconfiguration of the architecture meta-model, or in general any meta-level, must be managed to ensure system consistency and integrity at runtime (Moazami-Goudarzi, 1999; Blair et al., 2001; Wermelinger, 2000). In particular, ongoing computation and communication must be managed during reconfiguration, and this can be realised using a reconfiguration protocol.

A feature of existing approaches to architectural reflection for compiled programming languages is the requirement to explicitly specify the distributed system's software architecture using an architecture definition language (Allen, 1997; Cuesta et al., 2002b; Garlan and Schmerl, 2002; Moazami-Goudarzi, 1999). In K-Components, the architecture meta-model is constructed automatically and dynamically as components bind to one another and exchange software architecture descriptions. The architectural features reified in the architecture meta-model are the components, connectors, feedback states and actions in the system. Feedback states and actions are used by reflective programs to monitor and adapt components at runtime.

The causal connection between base-level and meta-level is a combination of an implementation link and a representation link. Reification of architectural events, including component creation/deletion, connector creation/deletion and connector binding/unbinding, from the base-level to the meta-level is implemented at intercession points in components and connectors. However, the reification of component feedback states is implemented via a representational link, providing a lesser form of consistency in the causal connection between the base-level and meta-level.

1.2.2 Adaptation Contract Description Language

In a system that supports architectural reflection, programmers can write reflective programs that encapsulate an application's self-adaptive behaviour by operating on the system's architecture meta-model. The reflective programs can reason about the state of components and connectors, perform adaptation actions on components and reconfigure the system by manipulating its architecture meta-model. In K-Components, an adaptation contract description language (ACDL) is provided to allow developers to declaratively specify reflective programs that encapsulate a system's self-adaptive behaviour. The ACDL separates the specification of a system's self-adaptive behaviour from the specification of its components' behaviour.

The term adaptation logic (Dowling and Cahill, 2001b) is used in this thesis to describe the policies, rules, goals or constraints that specify the self-adaptive behaviour of a system or component. In the ACDL, adaptation logic uses feedback states, defined on components and connectors, and feedback events to monitor components and connectors for conditions under which adaptation actions are triggered. Adaptation logic can be characterised as being reflective, as it reasons about the state of the system and can modify the system.

Adaptation logic can be specified using if-then rules or the event-condition-action (ECA) paradigm in the ACDL. This approach is useful where programmers can make reasonable assumptions about potential changes in the software's domain, are able to identify the events that are of interest, and how best to handle those changes i.e., what adaptation action to take. However, these approaches are unsuitable where the outcome of some adaptation action cannot always be predicted, such as in dynamic or uncertain environments, or where the space of possible events and adaptation actions is large. In such cases, there is a requirement for the online learning of adaptive behaviour. Collaborative reinforcement learning (CRL) is introduced in section 4.2.6 as a technique for the online learning self-adaptive behaviour and is supported in the ACDL.

1.2.3 Asynchronous Reflection

Existing adaptive systems built using reflective programming languages execute adaptation logic as reflective code at reification points in base-level code (Moreira et al., 2001) or as externally supplied operations (Garlan and Schmerl, 2002; Liu et al., 2004). Existing, compiled reflective programming models only support the execution of reflective code synchronously with program execution at reification points (Schaefer, 2001; Chiba, 1995), even though events that trigger adaptive or self-managing behaviour are often temporally orthogonal to program execution. Ganek claims that an autonomic system should be able to continuously monitor and adapt its components (Ganek and Corbi, 2003), but these existing reflective programming models do not provide first-class support for continuously executed reflective code. Native support to allow the continuous or periodic execution of reflective code in reflective programming models would help programmers meet Ganek's requirement for autonomic systems.

Asynchronous reflection is presented here as a technique that decouples the execution of a system's reflective behaviour from the execution of the system's base-level behaviour. Reflective code is encapsulated in autonomous programs whose execution is interleaved with the base-level program. In the

K-Component model, the base-level program is concerned with service provision and asynchronous reflective programs are concerned with the monitoring, planning and adaptation of the system. A performance advantage of asynchronous reflection over the traditional synchronous model is that it allows programmers to configure the execution-overhead of reflective code. In self-adaptive systems, this means that programmers can configure the trade-off between the extra overhead of executing reflective code and the responsiveness of the system to adaptation conditions. A system that supports asynchronous reflection can fulfil the requirement identified by Ganek of being able to continuously monitor the system i.e., components and connectors, for self-management conditions.

1.2.4 K-Components as Autonomic Components

The autonomic properties that can be supported by applications built using the K-Component model are constrained by both the set of states and events that can be monitored and the range of adaptation actions that can be performed. The K-Component model provides support for reasoning about the state of application-level components, but support for lower-layer events concerning middleware operation, the operating system and network adapters must be explicitly provided by wrapping legacy code as K-Components or reengineering the software as K-Components. The availability of middleware and lower-layer system events can help provide a more accurate model of an application's execution and communication context, and hence better support for building applications with autonomic properties.

The K-Component model supports different types of adaptation actions in the ACDL, including the dynamic reconfiguration of components and connectors by modifying an architecture meta-model and adaptation actions defined on components. Dynamic reconfiguration at the architecture level can be used to build self-optimising and self-healing applications by reconfiguring connections to faulty or poorly performing components. The ACDL does not yet support self-configuring or self-protecting adaptation actions, however, although this could be provided by extending the K-Component framework and ACDL compiler to generate code that makes use of discovery and security services.

1.3 Coordinating Self-Adaptive Components

Decentralised systems composed of autonomic components do not necessarily possess system-wide autonomic properties, as system-wide properties require coordination between components to enable components to take self-management decisions that are optimal for the system rather than for themselves. For autonomic distributed systems, Kephart notes that

> "System self management will arise at least as much from the myriad interactions among autonomic elements as it will from the internal self-management of the individual autonomic elements" (Kephart and Chess, 2003)

Much research in distributed systems has been concerned with building multi-agent systems that share control of a single model, i.e., a consensus-based approach to building distributed systems. In decentralised systems, however, it will not always be possible for agents to establish consensus over the current value of a variable. As a result, there has been a recent trend towards using decentralised algorithms and decentralised control techniques to engineer both large-scale and dynamic distributed

systems (Dorigo and Caro, 1999; Andrzejak et al., 2003; Ardaiz et al., 2003; De Wolf and Holvoet, 2003; Khare, 2003; Montresor et al., 2003; Boutilier et al., 2003), with areas such as peer-to-peer (Clarke et al., 2002) and ad-hoc networks (Curran and Dowling, 2004) producing noticeable achievements. A common pattern for decentralised system architectures is to model them as a collection of frequently similar, coordinating agents where each agent gathers information on its own, maintains a local, partial view of the system, takes independent decisions on how to behave and communication between agents is localised (Kennedy and Eberhart, 2001) or through some shared environment (Bonabeau et al., 1999).

A commonly cited example of a complex distributed system with decentralised control comes from the world of biology - the socially intelligent colony of ants (Bonabeau et al., 1999). Without any central co-ordination (since there is no ant in a social insect colony with the equivalent organisational power of a system architect!), the colony displays emergent behaviours (e.g., finding optimal foraging paths) and structures (e.g., organised piles of dead ants appear around the nest site). Individual ants do not have a global view of the colony and intelligent global behaviour and functionality emerges solely from local interactions (Bonabeau et al., 1999). The benefits of the decentralised approach to building distributed systems include the possibility of establishing system-wide properties such as self-regulation, self-configuration, self-optimisation, the lack of centralised points of failure or attack (Montresor et al., 2003), improved scalability as well as possible evolution of the system through evolving the local rules of the agents (Holland, 1996).

This thesis takes the view that large-scale[3] distributed systems with autonomic properties require *decentralised coordination* for their construction, with local coordination rules between components producing global autonomic properties. Coordination models can be found in multi-agent systems, but are primarily concerned with managing the inter-agent activities of agents collected in a configuration (Kielmann, 1996) and not with establishing system-wide properties. Examples of computer systems with decentralised control and coordination models where emergent system-wide properties are the result of locally specified rules can be found in Cellular Automata (Wolfram, 2002), self-stabilising distributed systems (Dijkstra, 1974), and bird flocking behaviour in Boids (Reynolds, 1987). A further requirement is that coordination models should be robust enough to establish system-wide autonomic properties in both predicted and unforeseen environmental conditions. This work investigates feedback as a specific decentralised control mechanism to build coordination models for self-adaptive components.

1.3.1 Feedback and Decentralised Coordination

In self-organising biological systems it has been discovered that

> "positive and negative feedback [are] the basic modes of interaction between components in self-organising systems" (Camazine et al., 2003)

Positive feedback is a mechanism that promotes changes in a system. When initial changes or fluctuations occur in a system, for whatever reason, positive feedback reinforces those changes in the

[3] On a scale comparable with autonomic systems from the world of biology.

same direction as the original fluctuation. Negative feedback, however, acts as a break or regulator on changes in the system that deviate from some optimal state.

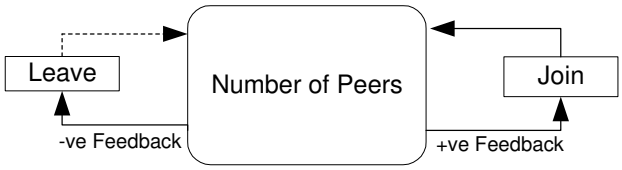


Figure 1.1: Feedback in the growth and formation of peer-to-peer networks.

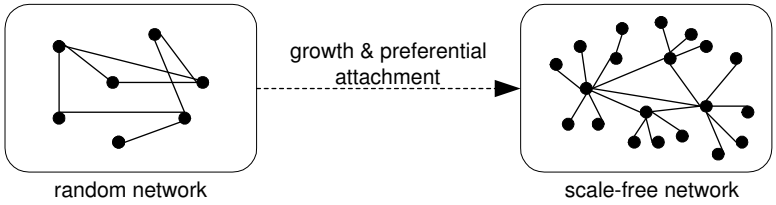


Figure 1.2: Positve feedback and the emergence of structure in peer-to-peer networks.

Examples of both positive and negative feedback can be seen in the formation and growth of P2P networks (see figure 1.1). As more peers join a P2P network more resources become available, and the attractiveness of the system to prospective users increases, resulting in even more peers joining the network. This phenomenon has also been called a network effect (Ripeanu, 2001). However, as more peers join the network, structural limitations of the network come into play and the performance of the network may degrade (Ripeanu et al., 2002). The poor performance of the system acts as negative feedback to both prospective and existing peers and leads to peers leaving the network. This negative feedback places a limit on the scale of the network.

Positive feedback can also cause the emergence of structure in systems. In the Gnutella P2P network a scale-free network topology emerged from an initially random topology (Ripeanu et al., 2002) through peers favouring certain (stable, high connectivity and high bandwidth) peers as neighbours over other candidates, in a process known as preferential attachment (Barabási, 2002). Gnutella shows how peers joining a network can reinforce changes in peer connection distribution, causing an unstructured (or random) network topology to self-organise into a scale-free network topology (see figure 1.2).

Feedback is a type of information flow in a self-organising system. There are different types of possible information flow for feedback in decentralised distributed systems. One type of information flow is direct communication where components send feedback information directly to one another. Another type of information flow is stigmergic communication where components do not communicate directly, but rather communicate feedback information to one another via a shared environment.

K-Components supports direct communication of feedback between connected components using connectors and by feedback events. Feedback events can also be communicated indirectly via shared K-Components to simulate stigmergic communication. On receiving feedback from a component, each K-Component takes an independent decision on how to update its view of the world and what adaptation actions (if any) to take. Typically when feedback is interpreted as negative feedback, it will trigger adaptation actions that serve to bring the value of some system state to within a desired

range, whereas feedback perceived as positive feedback will usually trigger adaptation actions that serve to increase changes in a system state in the same direction as the previous change.

The ACDL can be used to define coordination models between components based on positive and negative feedback. Decentralised coordination models can be designed to establish and maintain self-organising system-wide properties over groups of partially connected components. Coordination models based on feedback are the main building block for self-organising component-based decentralised systems with autonomic properties presented here.

1.3.2 Collaborative Reinforcement Learning

This thesis introduces a decentralised coordination model called *Collaborative Reinforcement Learning* (CRL) that establishes and maintains system-wide properties by solving decentralised optimisation problems. CRL is based on a goal-driven, unsupervised learning model called Reinforcement Learning (RL) (Sutton and Barto, 1998; Kaelbling et al., 1996).

In RL, an agent attempts to optimise its interaction with an environment by associating actions with system states. The 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 (Sutton and Barto, 1998). There is support in the ACDL specifying adaptation logic as a RL policy. States are represented as component feedback states, actions are represented as adaptation actions and reinforcements are calculated using a reward model provided by components.

The goal of a RL agent is to maximise the total reinforcements (reward) it 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 a 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 adaptive behaviour for the classes of distributed system that are intolerant to suboptimal action selection, such as real-time distributed systems.

CRL extends RL with a coordination model that is based on a variant of swarm intelligence algorithms (Kennedy and Eberhart, 2001) where agents interact locally with their neighbours and collectively learn from their successes. CRL does not make use of system-wide knowledge, and individual agents only know about and interact with their neighbours (Dowling et al., 2004). CRL can perform system optimisation over a group of decentralised agents for the purpose of establishing and maintaining system-wide properties. CRL solves system optimisation problems by specifying how individual agents solve discrete optimisation problems (DOP) using RL, advertise their results (their estimated cost of solving the DOP) to their neighbours, cache the results advertised by neighbours and delegate the solution to a DOP to a neighbour by initiating the start of a new DOP on a neighbouring agent (see figure 1.3).

CRL agents are designed to learn in dynamic environments where there is a requirement for continually updating both cached DOP costs advertised by neighbours as well as local environmental knowledge, such as neighbour availability. In CRL each agent's cache of estimated DOP costs for its

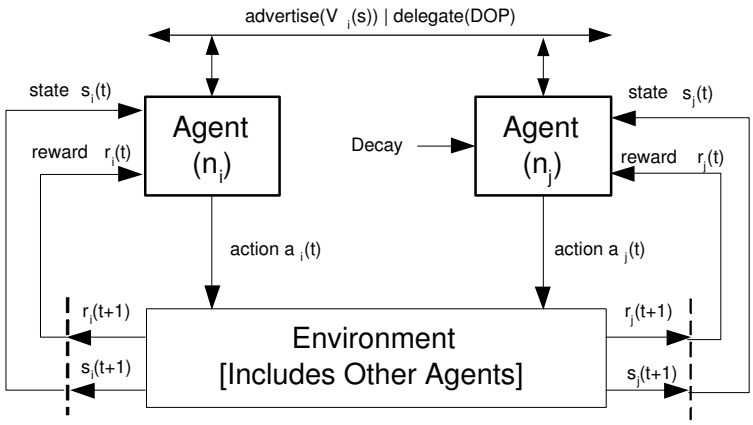


Figure 1.3: CRL agent-agent model.

neighbours decays over time to model the fact that such information is only valid for a certain period of time. Decay acts as negative feedback on cached costs in CRL. The rate of decay is configurable, with higher rates for more dynamic network topologies.

As a result of the decay model in CRL, there is a requirement for a continual flow of information in the system to maintain accurate knowledge of both system structure and estimated DOP costs. CRL supports the advertisement by agents of estimated DOP costs to neighbouring nodes as well as the initialisation of new DOPs by actors external to the system. Advertisement of DOP costs can provide positive feedback to neighbours and cause agents to converge on similar behaviours. Homogeneous components with converged local behaviours can perform the collective behaviour required for the establishment and maintenance of system-wide properties.

1.4 Contributions of Thesis

This thesis identifies self-adaptive components as building blocks for autonomic components and CRL as a mechanism for building decentralised systems with system-wide properties. Feedback models based on component and connector feedback states, as well as feedback events, can be used to build coordination models, such as CRL, that can enable the establishment of system-wide autonomic properties. The motivation for this work is to provide support for building decentralised systems with autonomic properties.

The main contributions of the thesis are the K-Component model, its support for building autonomic software and CRL for the construction of decentralised systems with autonomic system properties. The K-Component model contributes a programming model for building self-adaptive systems that includes a component model and an adaptation contract description language. It also includes a model of asynchronous reflection that enables the specification of adaptation logic that is executed asynchronously to program execution, as well as a model for learning self-adaptive behaviour online based on reinforcement learning. Different coordination models can be built using K-Components and CRL has been designed and implemented as a decentralised coordination model for K-Components.

The contributions are evaluated in the context of a decentralised file storage application that can self-optimise its storage behaviour by load balancing files at the system-level without the use of

global information, and automatically adapt and optimise its operation to changes in the system's environment, such as the introduction of a file storage server with increased storage capability. A self-healing property of the application at the component-level is also demonstrated by the automatic identification and reconfiguration of faulty connectors. The evaluation shows how CRL can enable an application to establish autonomic system properties without recourse to centralised techniques.

1.5 Towards Autonomic Distributed Systems

Given the previous definitions of an autonomic property of a system, an autonomic component and the lack of a widely accepted definition of an *autonomic distributed computing system*, we attempt to provide a definition here. Firstly, we consider a distributed computing system to be a decentralised system. Secondly, the autonomic properties of a decentralised system can only be externally observable, since its constituent autonomic components have partial views of the system. Although partial views can sometimes be representative of the system as a whole, we cannot make that assumption in the general case.[4]

We define an autonomic decentralised computing system as follows:

> "An autonomic distributed computing system has externally observable system-wide autonomic properties, that are established and maintained solely by the coordination and adaptation of its autonomic components that execute using only a partial view of the system, and without reference to the system-wide autonomic property"

1.6 Roadmap

The structure of the remainder of the thesis is as follows: Chapter 2 presents a survey of background material and related research in the field. It highlights the achievements and limitations of existing work in self-adaptive systems, as well as recent decentralised approaches to building distributed systems. Chapter 3 introduces the main concepts of the K-Component model, including the component model, architectural reflection, asynchronous reflection and the ACDL. Chapter 4 describes CRL, including how to specify a CRL policy in the ACDL. Chapter 5 presents the implementation of the K-Component model in CORBA. Chapter 6 evaluates K-Components as a building block for autonomic components and CRL as a decentralised coordination model for establishing autonomic properties using a decentralised load balancing application. Chapter 7 presents conclusions and future work.

[4] For example, in routing protocols for ad-hoc networks, traffic flows can only be observed by actors external to the system and not by the individual routing agents in the system (Curran and Dowling, 2004). Individual routing agents make local routing decisions using local information and do not use system-wide information to inform routing decisions. A routing agent cannot make assumptions about system traffic levels based on local, observed traffic levels.

Chapter 2

Background and Related Work

> "The point of philosophy is to start with something so simple as not to seem worth stating, and to end with something so paradoxical that no one will believe it."
>
> Bertrand Russell

This chapter introduces the problem domain and reviews existing self-adaptive and autonomic component models, as well as models for coordinating components in decentralised environments. A set of requirements for self-adaptive autonomic systems are developed and used as a guide to review a number of existing systems, each of which is representative of systems found in the different research areas that influenced the design of the K-Component model. A brief description of other related systems is also included. Finally, a comparison of the features and limitations of the reviewed systems is presented.

2.1 Coordination and Consensus in Decentralised Environments

The limitations of existing coordination techniques in decentralised systems provide the significant motivation for the development of CRL. We adopt Khare's model of a decentralised system as a system where every distributed agent manages their own local model of the system and the environment is characterised by uncertainty and dynamism. Analyses by Khare (Khare, 2003), van Renesse (van Renesse et al., 2003) and Montresor (Khare, 2003; Montresor et al., 2003) have identified the unsuitability of existing coordination techniques for decentralised systems, including centralised coordination models (Mikic-Rakic and Medvidovic, 2004) and consensus-based coordination models (Milner, 1999). Centralised coordination requires a single server that manages a global model of the system and the components in the system coordinate their behaviour using the global model. This technique is not viable in dynamic environments where access to the server is not always possible or decentralised environments where no global model of the system is available. In consensus-based models, a group of distributed components share control of a model. Techniques have been developed to ensure consistent updates to replicated copies of the shared model, e.g., group communication protocols. Consensus-based techniques, however, produce communication overhead when establishing agreement on changes to the shared model that limit their scalability (van Renesse et al., 2003) and their potential use in

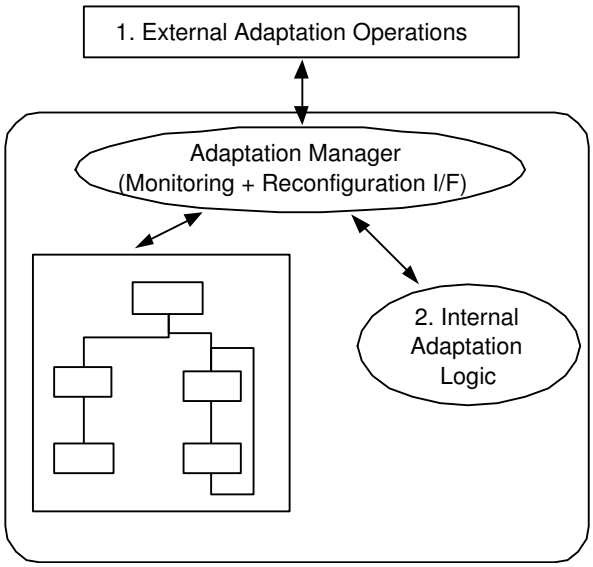


Figure 2.1: Internal and external adaptation using an Adaptation Manager.

building autonomic distributed systems in decentralised environments. Van Renesse characterises the limitations of applying consensus-based techniques to large-scale systems in (van Renesse et al., 2003):

> "traditional consensus protocols [...] have costs linear in system size [...]. With as few as a few hundred participants, such a solution would break down."

2.2 Adaptation and Autonomic Computing

According to Flake (Flake, 2000), the adaptation of a system is an internal process that is driven by changes in the system's external environment:

> "Adaptation is a feedback process in which external changes in an environment are mirrored by compensatory internal changes in an adaptive system."

As a computing system's environment changes over time, an autonomic computing system is presented with the challenge of optimally adapting its structure and behaviour to the new environment (Jann et al., 2003). The adaptation of software is driven by both predictable and unpredictable changes in the software's environment that naturally occur over time. Both longer-term requirements, such as the evolution[1] of software to meet new user requirements, and shorter-term requirements, such as the runtime adaptation of a system to overcome partial failures or sub-optimal operation, necessitate the adaptation of computing systems (Moreira et al., 2001). The ability to self-adapt to a changing environment is a fundamental property of autonomic computing systems that heal, optimise, protect and configure themselves. Over time, an adaptation process can require the constant regeneration of a system's structure.

2.2.1 Self-Adaptive Systems

It is important to make the distinction between adaptable and self-adaptive systems. Adaptable systems can be adapted to a particular deployment environment (Czarnecki and Eisenecker, 2000),

[1] Evolution has been described as a type of long-term adaptation process (Kennedy and Eberhart, 2001).

whereas self-adaptive systems adapt themselves to their operating environment[2] without manual intervention (Oreizy et al., 1999). Adaptable systems support their adaptation by an external actor (Wermelinger, 2000) using either a procedural or declarative interface (Blair et al., 2001). Self-adaptive systems, however, are subject to internal adaptation, triggered by monitored changes in either the system's internal state or the environment's state (Oreizy et al., 1999). Self-adaptive systems possess adaptation logic. Adaptation logic is the code that monitors a representation of the system's internal state and its environment and can perform conditional adaptation of the system in response to changes in system or environmental state. Adaptation logic is by nature reflective. Self-adaptive systems can be either closed or open dynamic systems. For closed dynamic systems, both the complete system behaviour and the behaviour that describes adaptations are specified at build time. Open dynamic systems, on the other hand, allow system behaviour to evolve after build time and are necessary if unanticipated adaptations are to be performed at run-time. Techniques for implementing open dynamic systems include dynamic linking mechanisms, dynamic object technology (including class loaders) and dynamic programming languages (Oreizy et al., 1999). Self-adaptive systems require no intervention from external actors, and can adapt themselves transparently to clients of the system, if desired. Figure 2.1 shows a schema of an architecture that supports both its external and internal adaptation.

Laddaga provides a definition for self-adaptive software in (Laddaga, 2000) that closely resembles the properties Ganek ascribes to autonomic components (see section 1). He defines self-adaptive software as:

> "Software that evaluates and changes its own behaviour when the evaluation indicates that it is not accomplishing what it is intended to do, or when better functionality or performance is possible."

2.3 Requirements for a Self-Adaptive, Autonomic System

Autonomic distributed systems require support for their runtime adaptation (Jann et al., 2003), e.g., in order to adapt the system to optimise the use of its resources (Nowicki et al., 2004). Systems adapt their structure or behaviour to a changing environment by performing *adaptation actions* (Sadjadi and McKinley, 2004). Adaptation actions should not affect the integrity of the running system. For a system that supports runtime adaptation to also be self-adaptive, it must also provide a decision making component that contains a decision policy that maps observed system states, describing the operating status of the system, onto adaptation actions. The effectiveness of decision policies should be both evaluated and updated over time in order to improve the self-adaptive behaviour of the system. Finally, in the case of system-wide adaptation of a component-based system, a coordination model is required to manage the collective adaptation of components, e.g., to maintain some system property. The minimum set of features a self-adaptive component model should provide in order to enable the

[2] A system's operating environment consists of the system's own software, the software subsystems that it uses (e.g., middleware, operating system, networking subsystem, etc), the software systems it collaborates with to perform its functions, and finally the and the system's users and their expectations of the software. This contrasts with the conventional understanding of the more general term environment in the domain of artificial intelligence, where the environment of an agent is everything that is external to the agent but can influence its behaviour or state.

construction of autonomic distributed applications for dynamic and uncertain environments is:

1. a set of system states whose values can be observed at runtime by a decision making component
2. a set of adaptation actions that can be executed by a decision making component to adapt the system at runtime, and indirectly its environment
3. mechanism(s) to ensure that adaptation actions maintain the integrity and consistency of the running system
4. a decision making component that associates adaptation actions with system states in a particular context using a decision policy
5. a mechanism or technique to automatically evaluate and update the decision policy over time
6. a decentralised coordination model to manage the collective adaptation of a group of components

2.3.1 State Monitoring

System state describes any runtime information that can be used by a software system to inform itself about its current operation. System state can be used as a basis for making adaptation decisions, but generally needs to be extracted, evaluated and abstracted prior to it being processed by the decision making component of the system (Hinnelund, 2004). This is due to the fact that raw system state may exist in many different formats and is found at many different levels in a distributed system, including within application components, at middleware-level, in the local neighbourhood or even distributed throughout the system. While an application will generally be able to monitor its internal state, system software must be instrumented to make its state information available to decision making components at higher levels. Research is ongoing into the problems of how to identify, acquire and model relevant state information in autonomic distributed systems (IBM, 2004; Birman et al., 2003).

System state information can also be used to build state-based models that describe system operation, i.e., the dynamic behaviour of the system. One potential problem with state-based models of large systems is that they are susceptible to the state explosion problem. For example, if we want to model the dynamic state of a system with state size n, when a new component containing m states is added to the system, the system state size increases to $m \times n$. However, in a decentralised system of components with equal state size, each component only needs to model its m internal states and $i \times m$ states for its i neighbouring components, where $i \ll n$. The number of states that each component models does not increase with system size. Decentralised systems offer the possibility of distributing the system's state space over multiple components in a network, offering the possibility of solving the state explosion problem.

Systems can improve their decision making capability by describing dynamic system operation using an abstract state model. The use of an abstract model of the system's state is also useful as it enables the execution of "virtual" adaptation experiments that may help predict the effect of executing an adaptation action (Whisnant et al., 2003). A state model can also make use of historical state as a basis for adaptation decisions, but a useful alternative to this approach, commonly used by control systems (Dutton et al., 1997), is to have system states satisfy the *Markov property* i.e., the state that

describes the system at a particular instance in discrete time is sufficient to determine all aspects of the future behaviour of the system when combined with knowledge of the system's future input (Barto et al., 1990). When system state satisfies the Markov property, the problem of evaluating the future state of the system after executing adaptation actions does not require historical information and, as a result, is more tractable. Finally, state-based models of the dynamic behaviour of a system, e.g., using time-dependent variables, enables the possibility of learning the dynamic behaviour of the system by formulating the learning problem as a search in the abstractly defined state space (Barto et al., 1990).

2.3.2 Adaptation Actions

A self-adaptive system needs a set of adaptation actions that can be executed to adapt the system at runtime to its changing environment. Adaptation actions can be *intrusive*, i.e., they can affect ongoing computation, communication or software dependent on the software being adapted, or *non-intrusive* in that they can always be safely executed concurrently with ongoing computation, i.e., they are either thread-safe or re-entrant. An example of an intrusive adaptation action is replacing a component in a software architecture that has existing clients, and an example of a non-intrusive adaptation action is a component changing its belief about the current operating state of the system. Intrusive adaptation actions generally operate on a representation of the system in a reconfiguration manager rather than directly on the system itself in order to meet adaptation consistency requirements (see section 2.3.3).

Adaptation actions can be performed at many levels in a distributed system, including the middleware (Duran-Limon and Blair, 2002; Atighetchi, 2003), component (Li, 2000; Moreira et al., 2001; David and Ledoux, 2003) and software architecture levels (Moazami-Goudarzi, 1999; Moreira et al., 2001; Garlan and Schmerl, 2002; Georgiadis, 2002; Dashofy et al., 2002). Depending on the system, it is often desirable for adaptation actions at lower levels to be transparent to higher levels (Almeida, 2001; Sadjadi and McKinley, 2004), such as making adaptation of middleware in response to changes in network connectivity, energy demands and security policies transparent to applications. In other cases it is more desirable for higher-levels to be aware of adaptations at lower levels (Blair et al., 2000), e.g., adapting an application to changing levels of QoS of network connections provided by the middleware (Sadjadi and McKinley, 2004).

Adaptation actions are *internal* to a system (Georgiadis, 2002) when the entity performing the adaptation is both co-located with and part of the same administrative domain as the entity being adapted. Adaptation actions are *external* to a system (Georgiadis, 2002) when the entity performing the adaptation is either a remote system or part of a different administrative domain. In heterogeneous distributed systems, components are not always part of the same administrative domain and security issues must be considered when allowing components perform adaptation actions on other components that reside outside of the original component's administrative domain (Khare and Taylor, 2004).

2.3.3 Adaptation Consistency

A set of system consistency requirements for adaptation actions were defined by Mozami-Goudarzi in (Moazami-Goudarzi, 1999), which state that a system is said to preserve adaptation consistency

requirements if:

1. the system meets the structural integrity requirements
2. the entities in the system are in mutually consistent states
3. the application state invariants are true

Firstly, the structural integrity requirements of a system determine the structure of a system in terms of the static relationships between its components and how components may be connected together. For example, the structural integrity requirements for replacing a component in the Rapide system (Luckham and Vera, 1995) require that the new component provides the same interfaces as the component it is replacing. Secondly, components in a distributed system must be in mutually consistent states in order to successfully interact with one another. Components are said to be in mutually consistent states if, after each interaction, the components involved make transitions to well-defined states. Most existing systems meet the mutually consistent state requirement by introducing the notion of a reconfiguration-safe state (Moazami-Goudarzi, 1999; Wermelinger, 2000) that must be reached before adaptation can occur. The reconfiguration-safe state should be reachable in finite, ideally bounded, time. In order to meet this requirement, many systems make assumptions about interactions finishing in finite or bounded time (Moazami-Goudarzi, 1999; Wermelinger, 2000). Thirdly, application state invariants are properties that hold over the components that make up an application. Some systems attempt to re-establish application invariants automatically (Moazami-Goudarzi, 1999; Almeida, 2001), often by transferring state from old to replacement components, while other systems leave the responsibility to the application developer.

Not all adaptation actions have the same adaptation consistency requirements. Non-intrusive adaptation actions have lower adaptation consistency requirements than intrusive adaptation actions, while intrusive adaptation actions may sometimes have different consistency requirements at runtime. Some systems provide support for specifying different implementation strategies for performing an adaptation action where an implementation strategy performs the same adaptation action with different adaptation consistency levels (Georgiadis, 2002). For example, adapting multimedia streaming bindings to changing QoS at the network-level does not require full adaptation consistency support (Blair et al., February 2000), as dropped frames that may occur during adaptation do not generally threaten the integrity of a multimedia application. Adaptation actions on e-commerce systems, however, would generally require support for full adaptation consistency.

2.3.4 Decision Policy

A self-adaptive system contains at least one decision making component that uses some decision policy to take adaptation actions based on observed system states and the "context" of the system, where context can be any external information used to inform the decision policy. The decision policy represents the adaptation logic in a self-adaptive system. The task of deciding on the optimal adaptation action to execute, given the system state and context is essentially a planning problem: how do we find an optimal decision policy for performing adaptation actions, given a (hopefully complete and correct) model of the system state, a set of available adaptation actions and a means

of evaluating the result of adaptation actions? The set of system states and adaptation actions determine, to a large extent, the ability of the decision making component to self-repair, self-protect or self-optimise a component or system's operation. In this thesis, context information is treated as regular system state information, internal or external to a system, used to inform the decision policy.

Many existing systems provide programming support for the separate specification of a system's decision policy. This helps achieve a separation between component programming and adaptation logic programming (Dowling and Cahill, 2001b), improving system maintenance and understandability. Different models that have been used to specify a system's decision policy include:

1. Decision trees (Liu et al., 2004; Roman, 2003)
2. Constraints in Architecture Definition Languages (ADLs) (Georgiadis, 2002; Moazami-Goudarzi, 1999)
3. Event-Condition-Action (ECA) paradigm (Efstratiou et al., 2002a; David and Ledoux, 2003; Adi et al., 2003)
4. Timed Automata (Blair et al., February 2000)
5. Finite State Machines (Technologies, 2002; Neema et al., 2002)
6. Fuzzy Control Model (Li, 2000)
7. Utility Function Policies (White et al., 2004)

The first five techniques are *action policies* that specify which action to take when the system is in a particular state, while a fuzzy control model produces a probabilistic policy for optimal action selection and utility function policies specify the relative desirability of alternative states in the system (White et al., 2004). Some self-adaptive systems attempt to define a *complete strategy* for the decision policy, i.e., specify all potential adaptation actions that could occur in all possible system states. Such a system contains the implicit assumption that it can predict both the outcome of an adaptation action and the future performance of the adapted system. However, defining a complete strategy quickly becomes infeasible as the space of possible system states and adaptation actions increases. For complex, instrumented distributed systems with N states and M possible adaptation actions, programmers cannot be expected to handle the $N \times M$ possible combinations or be able to accurately predict the outcome of executing some adaptation action.

Georgiadis distinguishes between 1st party and 3rd party (Georgiadis, 2002) decision policy implementations. In the "1st party configuration", the decision making functionality is a part of each component in the distributed system, while in the "3rd party configuration" the decision making component is an independent, external component. Decentralised, self-adaptive systems always have 1st party decision policies, as the dynamism of decentralised environments prevents external decision making components from maintaining consistent views of the components in the system. The decision making functionality in 1st party configurations can be scheduled to run asynchronously to computation in the components or, as in existing reflective systems, synchronously coupled with the execution of base-level operations on the component (David and Ledoux, 2003).

2.3.5 Evaluating and Updating the Decision Policy

In an autonomic or self-adaptive system, the result of adaptation actions should be evaluated to help understand the most effective adaptation actions to take given a particular system state and its history. The outcome and effectiveness of adaptation actions may be known instantaneously or not until some unknowable time in the future. Many existing systems require an administrator to manually evaluate the system's performance and update the system's decision policy using an administrator interface that allows policies to be plugged-in/out at runtime (David and Ledoux, 2003).

Examples of metrics used by existing systems to evaluate the performance of adaptation actions include:

- reconfiguration performance (Almeida, 2001). The aggregate system performance over a given time period after an adaptation action is performed can be used as a metric for evaluating the action. This metric requires knowledge of the expected system performance if no adaptation action were taken, and as such is less suitable for uncertain environments where this knowledge is generally not available.

- utilisation of system resources (Andrzejak et al., 2003; White et al., 2004). The difference in the utilisation level of system resources after performing an adaptation action can be used as a metric for evaluating the action. This technique is not viable for decentralised environments as it assumes the availability of global knowledge about resource utilisation levels.

The automated evaluation of autonomic system performance in decentralised environments is challenging as new metrics and techniques are required to evaluate system properties. How does a system evaluate its self-healing, self-optimising or self-protecting performance? This challenge can be addressed by a decentralised system that uses local performance metrics to learn a new, improved decision policy online.

Learning a Decision Policy

Learning a decision policy for adaptation actions provides systems with flexibility and robustness as it enables them to deal with uncertainty by updating decision policies to reflect a changing, uncertain environment. Learning is studied by computer scientists interested in developing software for domains that evolve and adapt over time. A wide range of techniques and mathematical theories for learning have been developed to handle tasks such as pattern classification, prediction, and adaptive control of dynamical systems (Sutton, 1988; Barto et al., 1990). In these techniques, learning is usually formulated as a search in an abstractly defined state space (Barto et al., 1990). Sequential decision making is a useful technique for evaluating the result of actions, whose effects have both short- and long-term consequences (Barto et al., 1990). Decision making tasks can be formulated in terms of a dynamical system whose behaviour emerges over time under the influence of a decision making component (Barto et al., 1990). Such decision making tasks can also be formulated as Markov Decision Processes. Stochastic dynamic programming (Barto et al., 1990) and reinforcement learning (Sutton and Barto, 1998) are widely used techniques for solving these tasks.

The goal of a self-adaptive system that learns its decision policy is to enable its decision making component to select adaptation actions that maximise the set of rewards accrued over a period of time, where a reward represents a scalar metric that quantifies the success of the adaptation action and is supplied by the decision making component's environment. The main challenge when designing such a system is *credit assignment* (Sutton, 1988). Given the state of the system and an action taken while the system is in that state, how can a system know whether that action was "good" or not in the context of delayed rewards*?* A decision making component should be able to take adaptation actions that produce sub-optimal rewards in the short-term but more optimal rewards in the longer term.

There may also be more than one decision making component in a system. Decentralised, 1st party decision making components can take adaptation actions on components that produce local rewards and learn a locally optimal decision policy over time. However, the decentralised decision making components may not produce a global decision policy that is optimal for the system, such as maximising utilisation of system resources. The problem of learning an optimal decision policy for a decentralised system is addressed in the next section on coordination models.

2.3.6 Coordination of Adaptive Behaviour

The problem of taking guaranteed globally optimal adaptation decisions in a distributed system requires complete information about the state of all components in the system and an optimal decision policy. A coordination model that implements the optimal decision policy can be data-driven or control-driven but requires support for strong consensus among components on the actions to be taken.

In existing dynamic software architectures the monitoring of system state and execution of adaptation actions is typically coordinated by a global configuration manager[3] (Allen et al., 1998; Wermelinger, 2000; Moazami-Goudarzi, 1999; Garlan and Schmerl, 2002; Georgiadis, 2002; Mikic-Rakic and Medvidovic, 2004), a centralised component in the distributed system that monitors component and connector states and coordinates the execution of adaptation actions on these components and connectors. The centralised configuration manager approach is not viable for decentralised distributed systems, due to its reliance on global state for the coordination of system adaptations.

A decentralised coordination model for adapting system behaviour requires the coordination of the self-adaptive behaviour of components without the use of global state or consensus-based distributed control. Decentralised coordination models cannot achieve strong consensus on the optimal adaptation actions to execute, but can achieve near-optimal decision policies through the localised coordination of components (Curran and Dowling, 2004; Jelasity et al., 2003). Decentralised coordination models allow components to share their local models with neighbouring components to reduce uncertainty about shared system states. Given a dense enough neighbourhood and accurate shared information, components can converge on common views of the system. Homogeneous components with converged local models of system state can perform the collective adaptive behaviour required to establish system-wide autonomic properties.

[3]Also known as a configurer (Georgiadis, 2002), configurator (Wermelinger, 2000), reconfiguration manager (Moazami-Goudarzi, 1999), architecture manager (Garlan and Schmerl, 2002) and deployer component (Mikic-Rakic and Medvidovic, 2004).

2.4 Techniques for Building Self-Adaptive Systems

This section reviews some well-known techniques for building self-adaptive software, including dynamic software architectures (Shaw and Garlan, 1996), reflection (Smith, 1984; Maes, 1987) and reinforcement learning (Kaelbling et al., 1996; Sutton and Barto, 1998). These techniques have been applied to existing self-healing and autonomic systems that can adapt themselves to changes in their underlying environment (Garlan and Schmerl, 2002; Blair et al., 2002; White et al., 2004; Whisnant et al., 2003; Hinnelund, 2004).

2.4.1 Dynamic Software Architectures

Dynamic software architectures have been used to explicitly model distributed systems as a reconfigurable graph of connected components. Architecture Definition Languages (ADLs) are typically used to specify a system as a directed, acyclic graph of components and connectors (Medvidovic and Taylor, 2000), with components as nodes and connectors as edges in the graph. In a software architecture, a component is an encapsulated, composable unit of computation and a connector is a first class entity that implements the interaction model for connected components. Component models generally specify the set of services a component offers via one or more provided interfaces as well as the services it uses as one or more required interfaces (Medvidovic and Taylor, 2000).

Architectural styles (Medvidovic and Taylor, 2000) or constraints (Moreira et al., 2001) are used to specify the properties of a software architecture (Georgiadis, 2002), and can be used to specify adaptation logic as rules for transforming the architecture from one configuration to another and the conditions under which the transformation is triggered (Moazami-Goudarzi, 1999). An architectural style is defined by Shaw as: "a set of design rules that identify the kinds of components and connectors that may be used to compose a system or sub-system, together with local or global constraints on the way the composition is done" (Shaw et al., 1995). Most existing architectural styles are consensus-based and use global constraints, making them inappropriate for decentralised systems. The adaptation actions supported by architectural styles are based on software architecture re-writing, i.e., the transformation of a software architecture from one valid configuration to another (Moazami-Goudarzi, 1999; Georgiadis, 2002). The reconfiguration of software architectures generally involves the execution of primitive reconfiguration operations at the level of a software architecture's components and connectors (Dashofy et al., 2002), such as `replace_component` or `rebind_connection`. There are different models for performing architectural adaptation (Moazami-Goudarzi, 1999; Wermelinger, 2000; Blair et al., 2001; Almeida, 2001) that meet the adaptation consistency requirements defined in section 2.3.3.

Dynamic software architecture approaches to building self-healing and autonomic software have been proposed by Schmerl and Garlan in (Garlan and Schmerl, 2002), Mikic-Rakic in (Mikic-Rakic and Medvidovic, 2004), Dashofy in (Dashofy et al., 2002) and White in (White et al., 2004). These approaches, however, are based on consensus-based architectural styles that use a global architectural reconfiguration manager, i.e., the architecture manager in (Garlan and Schmerl, 2002), the deployer in (Mikic-Rakic and Medvidovic, 2004), architecture evolution manager in (Dashofy et al., 2002) and the registry/sentinel in (White et al., 2004). A decentralised alternative to these consensus-based

architectural styles has been developed by Khare in (Khare, 2003) and is discussed in section 2.5.5.

2.4.2 Reflective, Self-Adaptive Software

Techniques such as reflection, e.g., object-oriented (Maes, 1987) or architectural reflection (Cazzola et al., 1999), have been used to automate the construction of meta models that can be used by a decision making component to adapt a system's structure or behaviour. In reflective systems, the meta models of the system are causally connected to the actual system objects/components are generally accessible via an interface to them called a Meta Object Protocol (MOP) (Kiczales et al., 1991). Meta objects can be adapted using the MOP and updates to the meta objects are reflected in the base objects. A MOP provides a system with openness that enables the examination of the state of a system and the modification of its structure or behaviour (Blair et al., 2002).

Reflective systems provide built-in support for the runtime manipulation of the system's causally-connected meta-model (Redmond, 2003). This allows the unconstrained reconfiguration of systems (Blair et al., 2002), e.g., architectural reflection supports the unconstrained reconfiguration of a system's software architecture, while reflective object-based systems support the unconstrained reconfiguration of their objects and object models (Redmond, 2003). Reflective systems require additional integrity management infrastructure to meet adaptation consistency requirements.

Reflective object-oriented programming languages have been used to build self-adaptive software (Dowling et al., 2000). However, reflective object-oriented programming languages were designed to open up a language's object model rather than build self-adaptive software. An implementation feature of existing compiled languages, such as Iguana/C++ (Schaefer, 2001) and OpenC++ (Chiba, 1995), is that reflective code can generally only be inserted and executed at reification points in the object model, such as object invocation or creation. When these languages are used to build self-adaptive systems (Dowling et al., 2000), the reflective code performing the adaptive behaviour is executed synchronously with program execution. This has the limitation of tightly coupling the execution of adaptation logic with the program execution, even though events triggering adaptive behaviour are often temporally orthogonal to program execution.

2.4.3 Reinforcement Learning

Reinforcement learning (RL) is a single-agent, unsupervised learning technique that has been used to build systems that can adapt and optimise their operation in an uncertain environment. Self-adaptive systems such as network routing protocols (Littman and Boyan, 1993; Peshkin and Savova, 2002), where each routing agent adapts its routing policy based on local information, and a traffic engineering application (Pendrith, 2000) have been built using RL.
In a typical RL model (Kaelbling et al., 1996; Sutton and Barto, 1998), an autonomous agent interacts with its environment by:

- observing the current *system state*
- selecting and executing one *action* from the set of available actions in that state
- observing the outcome of the action as a transition to a (possibly new) system state

- receiving a *reinforcement* as a scalar value that returns an immediate evaluation of the result of executing the action

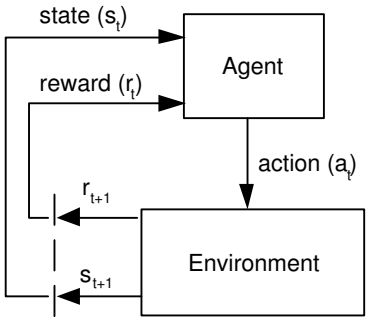


Figure 2.2: Reinforcement learning model.

Actions cause the system to make a state transition, and a reinforcement is received from the environment (see figure 2.2).

The goal of reinforcement learning is to maximise the total reinforcements 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 medium/longer term. Agents may also take actions in a trial-and-error manner in order to explore its environment for more optimal actions. 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.

The environment of an RL agent is usually modelled as a *Markov decision process* (MDP) (Kaelbling et al., 1996; Sutton and Barto, 1998). A MDP consists of:

- a set of states, $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$
- a set of actions, $\mathcal{A} = \{a_1, a_2, \ldots, a_M\}$
- a reinforcement function $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. The reinforcement is determined stochastically. $R(s, a)$ is the expected instantaneous reinforcement from action a in state s.
- a state transition distribution function: $P : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$, where $\Pi(\mathcal{S})$ is the set of probability distributions over the set $\mathcal{S}$. We write $P(s'|s, a)$ for the probability of making a transition from state s to state s' using action a.

The system may contain start and terminal states. An absorbing MDP is one where from every non-terminal state it is possible to eventually enter a terminal state. The goal of an RL agent that models its environment as a MDP is to optimise its interaction with its environment by learning an optimal policy.

Definition of an Optimal Policy

A decision policy (or policy) defines an action to take given the current state (Sutton and Barto, 1998), where $A(s)$ is the set of actions that can be taken given the current state s:

$$\pi : s \in S \rightarrow a \in A(s) \tag{2.1}$$

In a RL problem described as a MDP, a policy represents the behaviour of an agent, given the current state of the environment. The policy needs to address the credit assignment problem, so it considers the future consequences of each action, as well as their immediate outcome. The optimal policy, π^*[4], is the optimal set of actions to take from a given state, so as to maximise the reward accrued over a specified period of time. One such optimal policy is the discounted model that takes the long-term rewards into account, but rewards, r, received in the future are geometrically discounted according to discount factor $0 < \gamma \leq 1$. E is the sum of expected rewards (Sutton and Barto, 1998):

$$E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \,|s_t = s, \pi\right\}$$

This model allows a convergent sum with consideration of rewards into the future, while favouring rewards received in the near future. Given this model for expected future reward, a *value function,* V, is defined as the expected performance of an agent if it starts with a particular state and executes the policy (Sutton and Barto, 1998):

$$V^\pi(s) = E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \,|s_t = s, \pi\right\}$$

A *state-action function, Q,* can also be defined that calculates the expected performance of an agent if it starts in the state s and executes action a, and follows the policy thereafter (Sutton and Barto, 1998):

$$Q^\pi(s,a) = E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \,|s_t = s, a_t = a, \pi\right\}$$

The *optimal value function,* V^*, is defined as the expected performance of an agent if it starts with a particular state and executes an optimal policy (Kaelbling et al., 1996). For instance, using the infinite-horizon discounted model (Kaelbling et al., 1996), V^* is the policy that maximises the reward:

$$V^*(s) = \max_\pi E\left(\sum_{t=0}^{\infty} \gamma^t r_t \,|s_0 = s, \pi\right)$$

From dynamic programming (Bellman, 1957), it is shown that $V^*(s)$ can also be represented as:

$$V^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s')\right) \tag{2.2}$$

This is the solution to the *Bellman Equations* (Bellman, 1957). Note that by solving this equation, the solution does not require knowledge of the policy π. Maximising the reward (i.e., the right-hand side of 2.2) over the set of actions will follow an optimal policy.

An alternative representation for the optimal policy is the *optimal state-action function,* Q^*, that represents the value of executing a certain action and following the optimal policy thereafter:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \tag{2.3}$$

With this notation, $V^*(s) = \max_a Q^*(s,a)$. Thus, by maximising $Q(s,a)$ over the set of actions $\mathcal{A}$, an optimal action-value policy can be followed. The optimal action-value policy for an agent can

[4]The * indicates the policy is optimal

be found by solving the Bellman equations, although this is computationally expensive for a large number of states and/or actions.

Given a state model and a reward model for executing actions, the *optimal policy* $\pi^*(s)$ can be expressed using the optimal state-action function (Sutton and Barto, 1998):

$$\pi^*(s) = \max_a Q^*(s, a)$$

or using the optimal value function (Sutton and Barto, 1998):

$$\pi^*(s) = V^*(s, a)$$

Learning Strategies in RL

Reinforcement learning algorithms are used to update an agent's policy. RL strategies can be either model-free or model-based.

Model-based learning methods (Moore and Atkeson, 1993; Kaelbling et al., 1996) build an internal model of the environment, as a state transition model $P : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$, and calculate the optimal policy based on this model, whereas model-free methods do not use an explicit model and learn directly from experience. Model-based methods are known to learn in many settings much faster than model-free methods, since they can reuse information stored in their internal models (Appl and Brauer, 2002). In general, model-based methods have been less popular in RL because of their slower execution times and greater storage costs, especially as the state size grows. However, in distributed systems where acquiring real-world experience is expensive, the model-based approach has a distinct advantage over model-free methods as much more use can be made of each experience. Model-based learning requires that the state transition model and possibly the reward model are updated throughout the execution of the learning algorithm.

In many practical applications, however, either the state transition model is not fully known or the environment is non-Markovian. In these cases, the methods of temporal difference (Sutton, 1988; Sutton and Barto, 1998) and Q-learning (Watkins and Dayan, 1992) are able to obtain the optimal policy through executing actions and receiving rewards from the environment. Q-learning is a popular model-free learning algorithm that attempts to estimate the optimal policy using the function Q^*. The current estimate for the optimal policy is represented as $Q(s, a)$. When an action a is taken from state s, resulting in new state s' and reinforcement r, the Q-learning rule is used to update the policy:

$$Q(s, a) := (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') \right)$$

The parameter α represents a *rate of learning*. It can be proven that these $Q(s, a)$ values will converge to the optimal values $Q^*(s, a)$ if each action is executed sufficiently often in each state, and the α parameter is gradually reduced.

Distributed Reinforcement Learning

RL is a useful technique to build single-agent systems that can adapt their behaviour to optimise their operation in a static environment. As such it represents a promising approach to building systems that learn their autonomic behaviour. Distributed autonomic systems that wish to learn their autonomic behaviour require a distributed learning technique. Distributed reinforcement learning techniques have been developed to enable groups of independent agents to solve collective problems. Existing techniques use extensions to RL such as global reinforcement signals (Crites and Barto, 1998), shared global policies for agents (Mariano and Morales, 2000) and algorithms for coordinated reinforcement learning (Guestrin et al., 2002). While the first two techniques rely on centralised, global information, the coordinated reinforcement learning is a distributed technique that enables agents to coordinate on a shared representation of a policy by message passing. The algorithms, however, require strong consensus on a coordination graph of agents to determine globally optimal actions, and they have a "computational cost (that) is linear in the number of new 'function values' introduced" (Guestrin et al., 2002), meaning they are not application for large-scale distributed systems. The algorithms are also not suitable for decentralised environments, due to their requirement for strong consensus on shared state.

A more promising area of research for decentralised multi-agent RL systems has been the development of partitioned value functions. Schneider has investigated applying distributed value functions to a power grid (Schneider et al., 1999) and Stone applied Team-Partitioned, Opaque-Transition RL (Stone, 2000) to the RoboCup soccer problem. However, neither of these techniques address distributed systems specific issues such as changing availability of nodes and the cost of using network links. They also do not explicitly address the problem of dynamic environments, and how agents should adapt individually and collectively to changes in their environment. The development of distributed reinforcement learning techniques to solve system optimisation problems in decentralised distributed systems is an open research problem.

2.4.4 Analysis

Although it is also known that self-adaptive software can improve the effectiveness of its decision policy over time through the use of information relating to past adaptive behaviour, none of the existing self-adaptive component models provide support for exploiting such information.

The use of decentralised coordination models to manage the adaptive behaviour of components in a distributed system has not been addressed by existing self-adaptive component models or autonomic systems. In a decentralised system, self-adaptive components are limited to monitoring local state information and states in their local environment. In the next section we review previous work on coordination models for self-organising and decentralised software architectures by Georgiadis (see section 2.5.4) and Khare (see section 2.5.5) respectively.

2.5 Review of Existing Systems

The following sections of this chapter present detailed reviews of systems related to K-Components, with the reviews discussing how the systems meet the autonomic requirements presented in section 2.3. The systems reviewed were chosen either because of how their model for self-adaptive software can be used to build autonomic systems, how self-adaptive behaviour can be specified, updated or even learnt by the running system, and their ability to coordinate components in decentralised environments. The choice of the technologies and techniques used to build the systems was also a deciding factor, with reflection and declarative programming techniques influencing the inclusion of OpenORB (Blair et al., 2001) and QuO (Atighetchi, 2003), respectively.

The systems reviewed include a representative sample of the techniques from the different research areas that influenced K-Component's model of autonomic computing systems. OpenORB and QuO have concentrated on support for the construction of QoS-adaptive software where systems can adapt themselves to changes in the QoS provided by the underlying middleware or system. Accord is a recent self-adaptive component model targeted at building distributed autonomic applications and it is also reviewed. In the area of decentralised coordination of components, Georgiadis supports the construction of self-organising software architectures, while Khare provides an architectural style for decentralised environments. We also review Hinnelund's model of autonomic systems as control systems (Hinnelund, 2004) as it supports the learning of autonomic behaviour using machine learning. Finally, we present a short overview of other related systems.

2.5.1 QuO

Quality Objects (QuO) 3.0 is a CORBA-compliant middleware framework for building distributed applications that can adapt themselves to changes in their environment (Atighetchi, 2003). In particular, QuO was designed to support QoS-adaptive applications, e.g., multimedia applications that adapt the number of frames in a video transmission to the QoS of its network connection, but can be used to build other types of self-adaptive applications, such as the self-protecting application presented in (Atighetchi, 2003) that responds to events such as intrusion detection or TCP stack probes.

QuO supports the construction of self-adaptive client-server applications. Figure 2.3 shows the flow of control in a remote method invocation from a client to a server using the QuO middleware. Compared to a standard CORBA remote invocation using an ORB (Henning and Vinoski, 1999), it involves extra components, including a delegate object (that wraps the stub or skeleton object), contract object(s) that monitor state information encapsulated in system condition (syscond) objects and a mechanism/property manager that manages configuration information for the various components. The delegate and contract objects together encapsulate the adaptation logic for a QuO application. Although clients and servers can take independent adaptation decisions, system adaptation is consensus-based in QuO.

Quality Description Languages

QuO supports the separate specification of the adaptation logic for a client-server CORBA application using Quality Description Languages (QDLs). It also provides runtime objects that monitor middle-

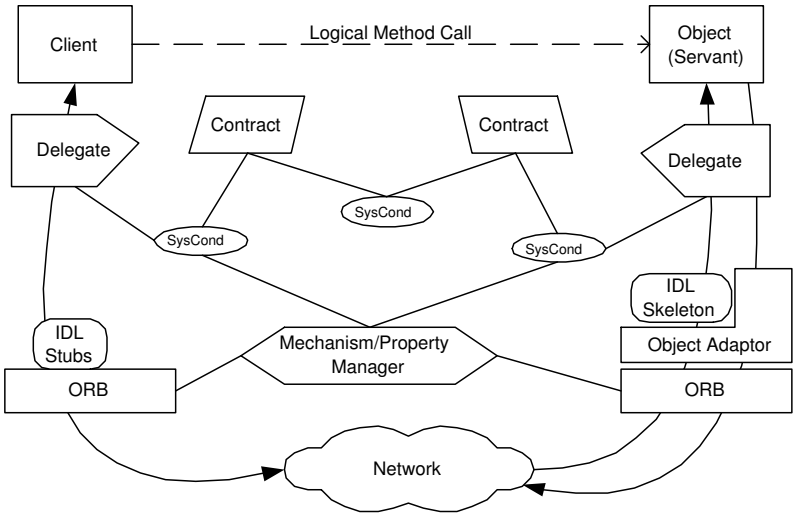


Figure 2.3: Client-server interactions in QuO middleware.

ware/system states and can be used in the QDLs (Technologies., 2002). The QDLs are the Contract Description Language (CDL) and the Aspect Structure Language (ASL). The CDL allows the specification of application-independent QoS contracts encapsulated in contract objects at runtime, while application-specific self-adaptive behaviour is specified in the ASL as QoS-adaptive behaviour and is encapsulated in a QuO delegate at runtime (see figure 2.3, adapted from (Technologies., 2002)).

A QoS contract in CDL specifies the QoS expected by the client, the level of QoS the servant expects to provide and adaptation actions to take if the QoS level moves outside the specified states. It allows the definition of QoS states (or regions) and adaptation actions to be executed in response to changes in QoS states. An application developed in QuO can provide a QoS contract that is either evaluated synchronously (in-line) with CORBA operation invocations using the delegate or asynchronously (out-of-band) using autonomous contract objects. A QuO contract consists of

- a set of nested states[5], each representing a possible state of QoS, where each state is either active or not
- a set of transitions indicating the adaptation action to execute on a state (or region) change
- references to system condition objects for measuring and controlling QoS
- callback objects that notify either the application-level client or the server object and are passed as parameters to the contract where they are used in transitions

As mentioned previously, the ASL is used to specify adaptation logic that is woven into delegates in a QuO client-server application. A delegate is a wrapper for a client-side proxy or server-side skeleton object. The delegate object intercedes in remote method invocations and executes adaptation logic that can monitor the state of QoS contracts and take adaptation actions such as making alternative method calls, calling alternative remote objects or invoking additional pre- or post-behaviour for a remote method. A delegate can only be generated for a CORBA interface in an IDL file(s) for which there is some specified adaptation logic in an ASL file(s) and where the dependent contracts are

[5]States were introduced in QuO 3.0 and can be used as an alternative to regions from earlier versions of QuO (Technologies, 2002). Regions can be modelled as states.

specified in a CDL file(s). The term advice, from aspect-oriented programming (Loyall, 1998), is used to describe adaptation logic and it can be inserted either *before*, *after*, *inplaceof* or *onexception* at the following join-points in delegates:

- methodEntry - the entry point where the method is called
- preMethodContractEval - the point where the state of the contract is evaluated before making the remote method invocation
- methodCall - calling of a remote method, i.e., call to the proxy or skeleton
- postMethodContractEval - the point where the state of the contract is evaluated after returning from a two-way remote method invocation
- methodReturn - when returning from the delegate method

QuO also provides a library of reusable QoS objects, called quosket objects, that can be used to instrument a client-server QuO application. Each quosket has an IDL interface, a contract describing the qosket's states, sysconds and adaptation logic, written in both the CDL and ASL, and some implementation code for the interfaces. Delegate objects can merge a quosket with application-specific adaptation logic, i.e., adaptive behaviour that is specific to the user of the QuO delegate. A code generator, quogen, is provided to compile both CDL and ASL specifications to either Java or C++.

State Monitoring

QuO provides feedback to an application about the state of the underlying middleware through system condition interfaces to resources and mechanisms. Sysconds monitor runtime information relating to the system's QoS and expose the state information they acquire using various interfaces to system resources, mechanisms and managers, including:

- ValueSC interface - store values for a syscond
- Probe interface - sysconds accept probing from the QuO kernel about its values
- Monitor interface - sysconds that poll in order to perform some action periodically
- SlidingWindowCounterSC interface - sysconds monitor the number of events that occur within a time window

Sysconds provide state information relating to the underlying system and middleware, and they can be grouped together to support the specification of application specific system state information that can be monitored at runtime.

The QDLs also support the specification of higher-level state information to abstract QoS values acquired from sysconds. In the CDL, programmers can define their own states and conditions (predicates) that determine state transitions (Technologies, 2002). The state clause syntax is:

```
state <name> [ until ( <predicate> ) ] ( <predicate> -> <state name>, ...  )
{ <nested regions or states> }
```

Predicates are Boolean expressions that consist of syscond object names and constants that are compared using relative operators (i.e., <, >, <=, >=, ==, <>, !=, and /=) and logical operators (i.e., and, or, and not). There is no support for application-level object state information in predicates. The `until` clause can prevent state transitions from occurring too rapidly, reducing hysteresis but at the cost of guaranteeing that the system is always in the correct state.

Adaptation Actions

Adaptation actions are supported as both application level operations and middleware or system level operations in the QDLs. QuO does not provide system support for intrusive adaptation actions, such as the replacement of server objects with alternative implementations. Developers must specify their adaptation actions in the QDLs.

Adaptation actions can be specified in the CDL as methods to be executed in state transition statements. The adaptation action is described in a transition statement body as a list of methods to be invoked when the state transition is triggered. Each method must be defined on a callback or a syscond object. Adaptation actions can be performed by synchronous or asynchronous decision policies. The contract waits for synchronous callbacks to finish before it can be evaluated again, while asynchronous callbacks are spawned as separate threads and the contract does not care whether they finish or not.

```
transition <from_region> -> <to_region> {
synchronous { <method_call> ...  }
asynchronous { <method_call> ...  }
}
```

Adaptation actions can also be specified in the ASL as advice. Advice is code that gets inserted at a pointcut in the delegate. ASL supports the following types of advice code:

- Function calls of the form, `<fn>(<args>)`
- Assignment statements of the form, `<var> = <expression>`, where `<expression>` can be any normal arithmetic expression, consisting of arithmetic operators, scoped identifiers, parentheses, constants, and function calls.
- Embedded Java or C++ code, of the form `java_code #{ ... }#` and `cplusplus_code #{ ... }#`, respectively.
- Exception Throw of the form, `throw <exceptiontype>`, where `<exceptiontype>` is an IDL defined exception.

The ASL supports the conditional execution of advice by associating advice with different states (or regions) in the delegate's associated contract. Here is an example region (or state) designator where different advice is called depending on whether the contract associated with the ASL, after evaluation, is in region A1 or region A2:

```
region A {
region A1 { <advice> }
region A2 { <advice> }
}
```

Adaptation Consistency

In the QDLs, it is the developer's responsibility to manage adaptation consistency for the callback and syscond methods that executed as adaptation actions. QuO does not provide system support for intrusive adaptation actions, and as such the existing published material on QuO does not explicitly discuss the adaptation consistency implications of adapting outgoing and incoming method invocations, e.g., by replacing a call to a server object with a method defined on an alternative server object. However, it is assumed that method invocation consistency (Almeida, 2001) can be easily maintained by delegates for actions such as server replacement as they have a single application-level thread that can safely wait for outstanding computation and communication to complete before calling a method on an alternative server object. Method invocation consistency meets the structural integrity and mutual consistency requirements, but QuO does not address the maintenance of application state invariants.

Decision Policy

As mentioned previously, the QDLs are used to specify an application's adaptation logic. QuO supports the execution of both synchronous and asynchronous 1^{st}party decision policies (Technologies, 2002). Synchronous policies are executed in the path of object interactions, while asynchronous policies are implemented as autonomous objects that monitor system or middleware state and execute adaptation actions independently of and asynchronously to object interactions. Both synchronous and asynchronous decision policies can be specified using the CDL, while ASL is generally used to specify application-specific, synchronous decision policies.

In the CDL, programmers can define a system's adaptation logic in a contract as a state machine, with conditions on state transitions and adaptation actions executed on a transition (Technologies, 2002; Neema et al., 2002):

```
contract <name> ( <args> ..)
{
<vars..>
state <name> <until_clause> ( <predicate> -> <state_name, ..} |
transition <transition_description> { <behaviour> ...} |
};
```

A contract can take both syscond objects and callback objects as parameters, in order to invoke methods on them as adaptation actions.

In ASL, the adaptation logic is specified as a behaviour description. A behaviour description associates a set of disjoint regions with adaptation actions. Only one region can be valid at any time. When a contract is evaluated it returns the current active region, and any adaptation actions associated with that region are executed.

Evaluating and Updating the Decision Policy

The performance and QoS provided by QuO applications can be evaluated online by system administrators using a GUI to the QuO kernel, (Java version only (Technologies, 2002)). The GUI provides frames to visualise the status of syscond objects that support the Probe interface and the active states

(or regions) of contracts. QuO C++ does not provide any support for the online updating of adaptation logic. To update a system's adaptation logic, a programmer has to write a new or modified QDL program, recompile, test and deploy the new application.

Coordination of Adaptive Behaviour

QuO supports the coordination of adaptive behaviour by allowing clients and the server to coordinate on a single, shared model of system state, e.g., by monitoring a shared syscond or quosket object. QuO applications are client-server based, and there is no support for the specification of a QuO application as a software architecture or a decentralised system. As such QuO provides applications with no system support for building coordination models based on distributed consensus or decentralised techniques.

Summary

QuO enables the construction of QoS-adaptive distributed applications. The specification of a system's decision policy is supported in the declarative QDLs. At the middleware and system levels, QuO provides syscond objects for monitoring system state. The execution of non-intrusive adaptation actions at the application and middleware levels is supported, but QuO does not provide system support for system reconfiguration adaptation actions.

QuO has many similarities with K-Components, including asynchronous monitoring behaviour that provides similar self-adaptive functionality to asynchronous reflection. The use of declarative contracts to specify adaptation logic is similar to K-Components but there is no support for monitoring application object state and the separation of adaptation logic from application objects is achieved using aspect-oriented programming techniques (Pal et al., 2000) rather than reflection.

QuO has several limitations that hamper its adoption for writing autonomic distributed applications. The development of QoS-adaptive applications is difficult as programmers have to familiarise themselves with many novel abstractions in both the CDL and ASL. Also, more complex self-adaptive applications quickly become difficult to specify and maintain as the number of states and adaptation actions grows. QuO also only supports the construction of client-server applications, and the updating of a system's adaptation logic requires system re-compilation and re-deployment. QuO is not suitable for the construction of decentralised autonomic applications due to its lack of support for building decentralised systems.

2.5.2 OpenORB v2 and OpenCOM

OpenORB v2 is a reflective, dynamically reconfigurable middleware platform built on top of OpenCOM (Blair et al., 2001), an open, adaptable implementation of Microsoft's Component Object Model (COM) (Microsoft, 2002) in C++. The goal of OpenORB is to support both distributed applications that can adapt to a changing middleware environment, by reflecting changes in the underlying middleware environment in the application's configuration, and also to provide a middleware that dynamically adapts its configuration to changes in the requirements and resource usage of its applications. OpenORB has the dual aim of adapting to meet application-level QoS requirements and improving overall middleware performance (Blair et al., 2003). The goal of maintaining QoS requirements, such

as timeliness or capacity requirements, in a changing and unpredictable network environment is similar to the autonomic computing goal of establishing and maintaining autonomic properties at runtime.

An OpenORB v2 middleware configuration is built using OpenCOM components and runs in a single address space. In OpenCOM, components implement interfaces, have outgoing connectors (called receptacles that represent explicit dependencies on other services) and are connected together at run-time. Outgoing connectors can be reconfigured at runtime. An OpenCOM address space provides a runtime for managing component configurations, and the runtime is used to manage OpenORB configurations.

Adaptation logic is provided in OpenORB v2 by Component Frameworks (CFs) that monitor middleware state information and can perform adaptation actions on OpenCOM components in a middleware configuration. CFs were defined by Szyperski as "collections of rules and interfaces that govern the interaction of a set of components plugged into them" (Szyperski, 1998). CFs in OpenORB encapsulate the monitoring code, adaptation actions and adaptation consistency management code for collections of related OpenCOM components representing different middleware configurations (see figure 2.5). CFs follow a meta-level manager/managed pattern with a CF adopting the manager role and its plug-ins adopting the managed role. A CF architecture is layered, with a higher-level CF being responsible for lower layers. In OpenORB, CFs can be added, removed and replaced by higher-level CFs, so long as this is allowed by the policies of the top-level CF. Examples of manager/managed CFs include ResourceManager/Resource CFs and TaskManager/Task CFs in (Blair et al., 2003).

State Monitoring

Both OpenCOM and OpenORB offer facilities for monitoring state information about a running system both in terms of its structure and its ongoing behaviour. The OpenCOM run-time provides facilities for introspecting the configuration of components in an OpenCOM address space. It provides different meta models to access information about the configuration of components, including the *IMetaInterception, IMetaArchitecture* and *IMetaInterface* models. Each OpenCOM component supports these interfaces as well as other component management interfaces, *ILifeCycle* and *IReceptacles*, for creating and deleting both components and connections at runtime. The *IOpenCOM* interface is a procedural interface for acquiring information from the OpenCOM run-time about its components and connections.

OpenORB v2 is built using OpenCOM and *IMetaInterception, IMetaArchitecture* and *IMetaInterface* act as a MOP to the underlying middleware configuration. *IMetaArchitecture* and *IMetaInterface* provide structural reflection for introspecting and modifying the underlying middleware architecture meta-model and interface meta-models for components, while *IMetaInterception* provides behavioural reflection (Blair et al., 2002) for introspecting and modifying behavioural aspects of components in the system (see figure 2.4). OpenORB v2 also provides separate support for the monitoring of component state in CFs. CFs that act as managers maintain management state information about their configuration of plug-ins and monitor events emitted by those plug-ins. Two different styles of QoS management component for monitoring state information supported by OpenORB are event collectors and monitors (Blair et al., 2002). Event collectors generate QoS events in response to observations

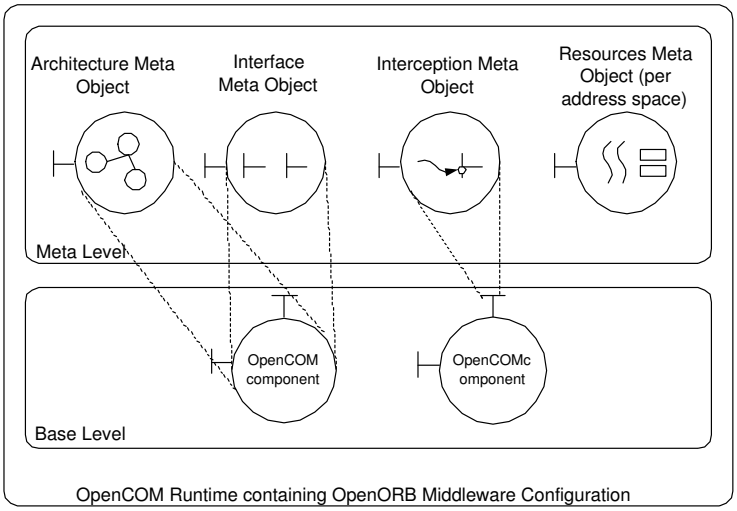


Figure 2.4: Meta models in OpenORB.

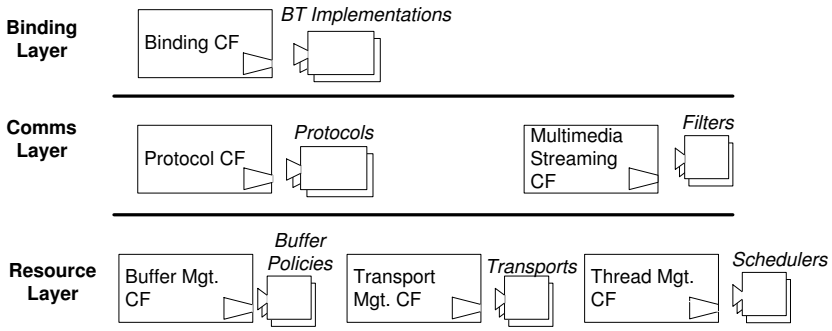


Figure 2.5: Component frameworks in OpenORB.

of underlying functional components, while monitors represent higher-level policy components that collect QoS events and report violations of QoS contracts to interested parties.

Adaptation Actions

In an OpenCOM address space, the *IOpenCOM* interface is a procedural interface that can be used for executing intrusive adaptation actions that reconfigure the system's software architecture, such as creating/deleting components and connecting/disconnecting components. OpenCOM also supports procedural interfaces, *IMetaInterception* and *IInterceptor*, that allow the creation and deletion of interceptors and their attachment/detachment to/from methods defined on interfaces.

At the OpenORB level, there is support for adaptation actions through both CFs and MOPs. These higher-level services use the lower-level OpenCOM adaptation actions when performing adaptations of the OpenORB middleware. In OpenORB, the *IMetaInterception, IMetaArchitecture* and *IMetaInterface* meta-level interfaces can be used to perform arbitrary manipulation of OpenORB's constituent components, although this form of unconstrained reconfiguration can easily violate the adaptation consistency of the system (Blair et al., 2001). As an alternative, CFs provide plug-in operations to add, remove and replace managed components from a running system that also maintain adaptation consistency. CFs constrain the scope of dynamic reconfiguration by providing different possible configurations and encapsulating rules for adapting configurations. In CFs, a management component provides access control, specified as layer composition policies at the top-level CF, to allow or disallow

a CF adaptation action on the basis of the current state information (layer composition information and meta-data) in the to-be-inserted CF. OpenORB provides two different patterns for management components in order to improve code reusability. Strategy selector components are designed to select an appropriate adaptation strategy based on the feedback they receive from monitors, while strategy activator components are reusable components that implement a particular strategy. Examples of CFs supported at the middleware level in OpenORB include a resource CF, a communication CF and the binding CF (Blair et al., 2001) (see figure 2.5).

Adaptation Consistency

At the OpenCOM level, adaptation consistency requirements for intrusive adaptation actions are met by providing per-receptacle locks that maintain method invocation consistency. When a reconfiguration of an outgoing connector is requested to the OpenCOM runtime, the runtime competes with threads performing operations on the connector for the lock on the connector. When the OpenCOM runtime acquires the lock, it can be sure that there are no outstanding invocation requests on the connector, ensuring mutual consistency, and can reconfigure the connector. When a component has to be deleted, the OpenCOM runtime acquires the locks for all connectors on the component before deleting it. Connectors are unlocked after the completion of reconfiguration actions.

However, when adapting OpenORB in an unconstrained manner using MOPs, the adaptation consistency requirements of the system may not be met (Blair et al., 2001). As mentioned previously, related groups of components are managed as configurations in CFs. CFs encapsulate the adaptation actions and adaptation consistency management code for adapting collections of related OpenCOM components that make up valid middleware configurations. A layer composition policy controls reconfiguration of a top-level CF. This layer composition policy is implemented as a component that validates proposed layer composition operations, such as the insertion or removal of CFs and can help maintain structural integrity when dynamically reconfiguring OpenORB.

OpenORB supports the specification of QoS-adaptive applications, and can thus help maintain application state invariants specified as QoS requirements. OpenORB does not, however, provide support for the automated transfer of state between old and replacement components to maintain application state invariants and requires application developers to handle this problem.

Decision Policy

OpenORB v2 supports the development of 1^{st}party decision policies as CF management components provided by an OpenORB developer. There is no declarative programming support for decision policies in OpenORB v2, although the previous version, OpenORB v1, provided a scripting language for the specification of QoS management logic as timed automata (Blair et al., February 2000). In OpenORB v2, monitors and strategy selector management components encapsulate the higher-level policy for adapting an OpenORB configuration, while event collectors and strategy activators encapsulate the lower-level mechanisms for monitoring and adapting the CF (Blair et al., 2002). The management components act as timed automata interpreters at runtime (Blair et al., 2002).

Evaluating and Updating the Decision Policy

The decision policy in a management component in an OpenORB system is open to inspection and adaptation through reification of the management components, providing a kind of reflective tower (Blair et al., February 2000). The authors discuss in (Blair et al., February 2000) how a meta-manager could be developed by monitoring the behaviour of the management component to detect the number of adaptation actions performed by the management component.

An OpenORB application's decision policy can be updated at runtime by dynamically inserting and removing management components in a CF. Managed components provide events about system state and do not need to know in advance that they will be managed, enabling new management components to be introduced at runtime without affecting system integrity.

Coordination of Adaptive Behaviour

OpenORB supports limited coordination of the adaptive behaviour of clients and servers. The binding framework supports coordination on the adaptation of client and server binding types to ensure consensus on the updated binding types to be used by clients and servers running on different OpenORB instances. OpenORB applications are client-server based, and there is no support for the specification of an OpenORB application as a software architecture or a decentralised system. As such OpenORB provides applications with no system support for building coordination models based on distributed consensus or decentralised techniques.

Summary

OpenORB v2 is built using OpenCOM and supports the construction of self-adaptive distributed applications/middleware systems using reflective techniques. It is representative of a larger class of reflective, self-adaptive systems and also of many QoS-adaptive systems. OpenORB supports the specification of a system's adaptation logic using a CF model that monitors system states and executes adaptation actions that maintain adaptation consistency in conjunction with OpenCOM. The decision policy is realised as a management component that can be introspected and updated by an administrator at runtime.

OpenORB applications can be engineered to provide many autonomic properties at the client-server level, such as self-healing (Blair et al., 2002), as self-adaptive system behaviours. Similar to QuO, however, the applications will become difficult to specify and maintain as both the number of system states to monitor and number of adaptation actions that can be performed increases. OpenORB is not suitable for building decentralised, autonomic systems due to its client-server centric model and the lack of support for building decentralised coordination models.

2.5.3 Accord

Accord is a recent component-based programming model and framework, designed to support the development of autonomic applications (Liu et al., 2004; Liu and Parashar, 2004). The framework allows the development of "autonomic components"[6] (Liu et al., 2004) and distributed systems with

[6] The authors do not provide a definition for an autonomic component.

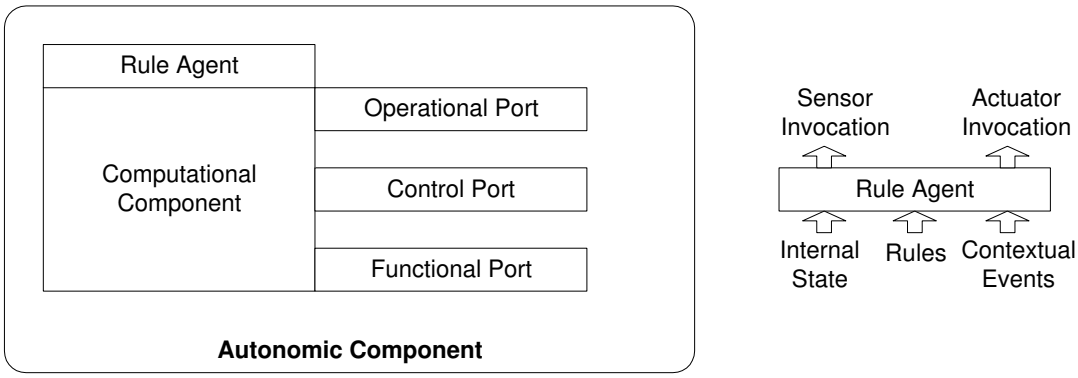


Figure 2.6: Autonomic component model in Accord.

autonomic properties are built by dynamically composing and managing components. The goal of Accord is to enable the organisation, interaction and coordination of autonomic components in a distributed system that can be managed at runtime using high-level rules. The authors identify challenges inherent in the construction of large-scale autonomic distributed applications including heterogeneity and dynamism in the availability and state of components, services and infrastructure. They also note the presence of uncertainty due to dynamism, failures and incomplete knowledge of the system.

The Accord programming framework attempts to support the construction of distributed applications that meet these autonomic challenges and are able to manage themselves using minimal human intervention. Accord is built on the AutoMate middleware (Agarwal et al., 2003), a middleware for self-managing Grid applications. The four main concepts that make up an autonomic application built using Accord are:

- an application context that describes a common semantic view of the application
- the autonomic components
- a set of rules and mechanisms for the dynamic composition of autonomic components
- an agent infrastructure to support rule enforcement that realises the dynamic composition and self-managing behaviour

Figure 2.6 shows the structure of an Accord component with its three types of ports: a functional port, a control port and an operational port. The functional port describes the operations provided by and used by a component. The control port is a set of tuples (σ, ξ) where σ is the set of actuators and sensors in the component and ξ is the constraint set for the sensors/actuators that uses state, context information and policies to determine when and how the sensors/actuators are accessed and by whom. The operational port allows the dynamic insertion/removal of rules specifying adaptation logic and manages existing rules.

Rule agents are assigned to each component and manage the component by monitoring its state and context, and storing rules (adaptation actions) that can be fired to invoke operations on the component via the three ports. Two types of rule supported in the rule agent are behaviour rules that define the functional behaviour of the components, and interaction rules that coordinate the interactions

between components, and between components and their environments. All binding and interactions between components are managed using interaction rules instead of the more traditional proxies or connectors found in many existing distributed component models. This produces components that do not make independent decisions about which components to connect to, as their binding information is encapsulated in their associated rule agent.

Rule agents can also cooperate to fulfil application objectives and interact with each other using a decentralised reactive tuple-space from the coordination infrastructure for Accord called Rudder (Liu and Parashar, 2004). The reactive tuple-space provides context-aware agents that manage components, monitor and analyse system state, support agent coordination, the execution of composition rules and the creation of application workflows. Tuple-space-specific behaviours can be programmed using the event-condition-action paradigm.

Accord supports the composition of autonomic components as "an organisation of components and the interactions among these components" (Liu et al., 2004) based on the composition of functional ports (provides and uses interfaces). Dynamic composition of systems in Accord involves delaying until runtime decisions about which components are composed together and how and when they interact. The composed components that make up an autonomic application are described in a workflow graph where both nodes (components) and edges (interactions) can be added, removed and updated dynamically. This is similar to how dynamic software architectures model a system as a configuration graph with nodes as components and connectors as edges. A composition agent manages dynamic composition by maintaining global knowledge of the workflow and decomposing the workflow (configuration) into a set of interaction rules that are injected into the appropriate rule agents. The rule agents then execute the rules to configure their components and establish their interaction relationships.

State Monitoring

Accord supports the monitoring of component states using sensors defined in the control port of a component. Agents in Accord are able to monitor component states by polling a component's control ports. No abstract representation of a component's state model is presented in existing publications (Liu et al., 2004; Liu and Parashar, 2004). Similarly, there is no available model for the tuple-space described in Rudder (Liu and Parashar, 2004; Li and Parashar, 2004).

Adaptation Actions

Actuators in the control port of components can be used to modify the state of the component at runtime and can be used by agents to dynamically adapt components. Composition agents can adapt the set of components in an application workflow. This can involve using rule agents to change interaction rules and adapt the connections between components or between components and their environments. Accord supports the following intrusive adaptation actions, executed and managed by composition agents and rule agents:

- the addition/deletion/replacement of components in workflows by a composition agent;

- establishing/deleting/changing interaction relationships by rule agents and managed by composition agents.

Adaptation Consistency

Accord only allows the replacement of a component with a component whose functional ports are type compatible, thus maintaining structural integrity requirements. The reconfiguration process is managed by a composition agent that ensures that the component being replaced is in a reconfiguration-safe state before it is replaced, thus ensuring mutual consistency requirements. On reconfiguration, the old component transfers its rule set to the new component and notifies dependent components to update their interaction rules. This helps maintain application state invariants concerning the component's interactions. There is also the possibility that multiple rules will be triggered concurrently, so in Accord conflicting rules are resolved by associating a priority with rules, where low priority rules are forced to wait for locks held by rules with higher priority.

Decision Policy

The different agents in Accord, e.g., configuration and context agents, encapsulate decision policies for monitoring and adapting managed components. Configuration agents are centralised entities that encapsulate a 3^{rd} party decision policy for an application workflow (Liu and Parashar, 2004; Li and Parashar, 2004), and they interact with the rule agents for the components in the workflow to perform reconfigurations. Components also contain a 1^{st} party decision policy in the form of interaction rules in the rule agent. The interaction model for components defines how and when components interact. They are specified as a set of `if-then` statements (Liu and Parashar, 2004; Li and Parashar, 2004), with conditions being receipt of messages or predicates on component state and adaptation actions being operations on a port on the local component or the sending of a message to a remote component. An example of a rule in (Liu et al., 2004) is

```
C1 IF terminationMsg is received THEN invoke stop;
C3 IF isResourcedBalanced()==false THEN send loadMsg to DSM;
```

where DSM is a remote component. In the interaction rule C1, a local `stop` operation is invoked if a `terminationMsg` is received from a remote component. In the interaction rule C3, a `loadMsg` is sent to the remote component DSM if a local method `isResourceBalanced()` evaluates to `false`.

Evaluating and Updating the Decision Policy

The interaction relationships for components in a workflow can be updated at runtime by external components, such as a configuration agent, inserting new interaction rules dynamically to rule agents. There is no mechanism provided for the automated evaluation and updating of interaction rules at runtime, and Accord requires an administrator to manually evaluate the behaviour of the system and insert/remove/replace the appropriate interaction rules using the configuration agent.

Coordination of Adaptive Behaviour

In Accord, the centralised coordination of the adaptation of workflows is supported by a composition agent. Accord, through Rudder, also supports the definition and execution of decentralised coordination models specified as programmable reactive behaviours in a tuple-space. A reactive tuple consists of a condition for triggering some reactive behaviour, the reactive behaviour itself and a guard for defining the execution semantics of the reactive behaviour, e.g., execute immediately and once. There are no decentralised coordination models presented, however.

Summary

The Accord model is designed to support the construction of autonomic applications, such as Grid applications, in closed, distributed environments. The use of centralised composition agents to bootstrap the dynamic composition model is a reasonable approach for a closed system, where all components are under the same administrative domain, but it is not viable for heterogeneous, decentralised systems. Aspects of the model resemble dynamic software architectures where workflows are equivalent to configuration graphs and interaction rules provide similar functionality to connectors with encapsulated binding information.

The accord model is representative of the existing autonomic component models, including IBM's autonomic component model (Kephart and Chess, 2003). Similar to other autonomic component models, much of the work on Accord is early in development. There is no definition provided for an autonomic component and Accord's context model has not been specified (Liu et al., 2004; Liu and Parashar, 2004). In addition to this, the languages used to specify components, the interaction rules and the workflow configuration have not been defined in published material. The implementation of Accord is also at an early stage, and at the time of publication of (Liu et al., 2004) the dynamic composition model had not been implemented. The reactive tuple-space model offers the potential of building decentralised autonomic applications, but it does not support any decentralised coordination model yet.

2.5.4 A Self-Organising, Consensus-Based Software Architecture

Georgiadis, Magee and Kramer introduced the notion of self-organising software architectures for distributed systems in (Georgiadis, 2002; Georgiadis et al., 2002). They define a self-organising software architecture as "one in which components automatically configure their interaction in a way that is compatible with an overall architectural specification" (Georgiadis et al., 2002). Rather than define a software architecture as a configuration of components using an ADL, a self-organising software architecture is specified using local architectural constraints on how components are composed, rather than the more usual global constraints. Component state is also used to influence self-organisation via attribute objects defined on components.

The component model is heavily influenced by Darwin (Magee et al., 1995), a software architecture model. In the component model, the provided and required interfaces of each component are managed by Port objects. Each component contains a component manager that stores local architectural constraints that should help achieve the desired global architectural style. The component

manager manages its component via its Port objects and maintains a local view of the state of the software architecture configuration. When a port on a component in the software architecture is bound/unbound to another component, an event is generated. These events are broadcast to all component managers in the network that independently update their view of the system configuration. The totally ordered delivery of these events provides distributed consensus on the current view of the system configuration at any instant in time. On receipt of a component insertion/removal event or a component attribute change event, every component manager in the system evaluates its local system configuration view against its local constraints. This may result in the execution of adaptation actions, such as binding/unbinding components, to satisfy its constraints.

State Monitoring

The state of the software architecture configuration is monitored by component managers using events generated by ports binding/unbinding to/from components. Each component manager maintains a local model of its view of the software architecture configuration in the form of a directed graph of components and a descriptor for each component type in the graph (Georgiadis, 2002).

Adaptation Actions

The self-organising software architecture model supports intrusive internal and external adaptation actions by component managers. Internal adaptation actions are limited to `bind/unbind/rebind` actions that operate on required ports on a component manager's local component. External adaptation actions involve the notification of changes to the software architecture configuration graph to other components, either as a result of component addition/removal, attribute changes on components or because of internal adaptation actions. The external adaptation actions presented in (Georgiadis, 2002) are `join/leave/attrib` adaptation actions that operate on components. External adaptation actions update the local view of the software architecture at the components in the system.

Each component manager independently executes adaptation actions to maintain its local architectural constraints. A selector function implements the architectural constraints as a constraint satisfaction algorithm:

$$selector(p) : G \xrightarrow{a_i^p} G'$$

where actions a_i^p are internal adaptation actions (`bind/unbind/rebind`) involving a port p at node i, and G is the configuration graph for the software architecture that is re-written after the adaptation action.

Adaptation Consistency

In order to maintain adaptation consistency in the distributed software architecture, the self-organising architecture has to provide adaptation consistency for both internal adaptation actions and external adaptation actions. Georgiadis provides atomic adaptation actions: actions that either succeed, resulting in the modification of the state of all components involved in the adaptation, or fail, in which

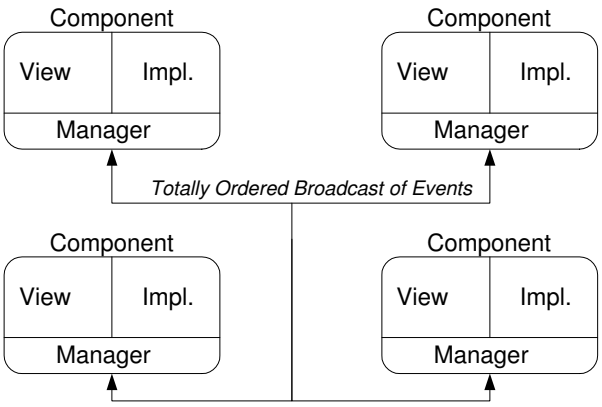


Figure 2.7: Updating configuration views.

case there are no state changes in any component. Furthermore, adaptation actions result in the execution of external adaptation actions that broadcast all state changes to all component managers that then update their local views. In order to maintain adaptation consistency of these external adaptation actions, a totally ordered broadcast scheme is used. There is no system support for the maintenance of application state invariants.

Decision Policy

The model supports a 1^{st}party decision policy described by a set of configuration rules that capture local architectural constraints as: `<required-port, selector, action-list>`. The `required` port is the port to which the rule applies, `selector` functions are associated with ports and contain the adaptation logic that is executed when a configuration view changes, and finally the `action-list` is a set of available internal adaptation actions. In the current implementation, there is no declarative programming support for specifying configuration rules and adaptation logic can be provided as Java classes.

Evaluating and Updating the Decision Policy

There is no support for the evaluation of the decision policy. Some limited support for updating the decision policy dynamically is provided by the ability to dynamically update adaptation logic encapsulated in load selector functions using Java's dynamic class loading capabilities. This facility, however, does not provide guarantees of minimal disruption to the system.

Coordination of Adaptive Behaviour

The coordination model is an example of a consensus-based architectural style (Khare and Taylor, 2004) where local changes are atomically broadcast to ensure that all components have correct local views of the system architecture (Georgiadis, 2002). This impacts, however, on the scalability of the system and is not a viable approach for decentralised systems (see section 2.1).

Summary

Although the work presented here is a software architecture that organises itself through local architectural constraints, it does not provide a model for building decentralised systems, as the coordination model is consensus-based, requiring components to broadcast local changes to all other components in order to maintain common views on the system. Additionally, there is the assumption that network partitions do not occur, since the atomic broadcast mechanism is used to maintain a consistent view of the architecture. There is also the problem of stabilisation on architecture configurations, since after each component manager receives an event indicating that the configuration view has changed, they may individually decide to issue more reconfiguration requests to satisfy their architectural constraints. These reconfiguration requests may generate cascading reconfiguration requests from component managers. There is no guarantee that reconfiguration requests will stop at any future moment in time.

The self-organisation of a software architecture to preserve system-wide architectural properties is similar to how autonomic systems could self-organise to preserve system-wide autonomic properties (Minsky, 2003). Georgiadis' model, however, is not suitable for the construction of decentralised autonomic systems as it uses consensus-based coordination techniques to preserve architectural properties.

2.5.5 A Decentralised Software Architecture

Khare attempts to cope with uncertainty and disagreement in distributed systems by adopting a decentralised approach to specifying software architectures that uses local constraints on configurations of components and connectors to induce desired system-wide properties (Khare and Taylor, 2004). An example of an induced property in a decentralised system is near optimal estimates by agents of global or shared variables. Khare's model for decentralised software architectures is motivated by temporal and trust problems associated with establishing consensus in large-scale, open distributed systems.

He introduces new capabilities to Representational State Transfer (REST), an architectural style used to describe the World Wide Web (Fielding, 2001), in order to support the construction of decentralised systems, including asynchronous event notification, a routing proxy to support independent extension of components owned by 3rd parties, enforced serialisation of updates to a resource, decision functions to select the current value of a shared resource and techniques to estimate current representations based on past ones (Khare, 2003; Khare and Taylor, 2004). The styles are combined to present a new architectural style for decentralised systems designed to handle uncertainty found in distributed systems, including uncertainty due to message loss, network congestion, message delay and disagreement.

Khare's decentralised architectural style allows independent agents to make their own non-intrusive adaptation decisions based on local estimates of shared views or states of the system, instead of attempting to achieve consensus (or simultaneous agreement as he calls it) on the state of the system. In his model of a consensus-free system, the true value of a global view or state is replaced by local estimates (or opinions), due to the fact that the actual value cannot be known. As a solution to this problem, the model has an approach described by 'BASE' properties: to "rely solely on Best-effort network messaging; to Approximate the current value of remote resources; to be Self-centred

in deciding whether to trust other (agents') opinions; and Efficient when using network bandwidth" (Khare and Taylor, 2004). Components use these properties to try to establish a best estimate of the value of a global state.

The model introduced in his thesis, however, does not support the construction of self-adaptive or autonomic software. The prototype system developed using Khare's model does not demonstrate the kind of self-adaptive behaviour described in the previously reviewed systems. It is an Automarket car auction system that demonstrates how agents in the system can provide local estimates of the global auction price of vehicles and adapt their interpretation of the types of vehicles that are for sale.

State Monitoring

Instead of using references to shared or global state, Khare uses end-to-end estimator functions to monitor the value of a shared variable/resource/structure. The estimator functions are decentralised decision functions that estimate the value of the shared state. In order to reduce the uncertainty of the estimator functions, he proposes that local agents use both their own estimate of the value of the shared state and the opinions of several different agents to improve the estimated of the value of the shared state. In his model in (Khare, 2003), estimator functions are modelled as simple caches at different levels in the system. The estimator function uses local values in the cache to estimate the value of some remote state.

Adaptation Actions

The adaptation actions presented in Khare's example system in (Khare and Taylor, 2004) are non-intrusive adaptation actions that are limited to setting the price of goods in a car market. There are no intrusive adaptation actions, such as architectural adaptation actions, presented in his model.

Adaptation Consistency

As Khare's model does not support intrusive adaptation actions, there are no adaptation consistency requirements.

Decision Policy

The 1^{st} party decision policy presented in (Khare and Taylor, 2004) is not based on associating a particular adaptation action with system state, but rather with estimating the value of a shared resource. Decentralised decision functions may use components such as a Predictor component, to predict the current state from past data, or an Assessor component, to allow agents to collaborate when estimating the value of a shared variable, to adapt estimates. Detailed discussion of implementation strategies for these components or how to adapt estimates is lacking in (Khare, 2003; Khare and Taylor, 2004).

Evaluating and Updating the Decision Policy

Khare suggests that a predictor component can be used to learn how to estimate the value of a shared resource or variable, although he doesn't present details on how such a predictor component would

work.

Coordination of Adaptive Behaviour

Khare abandons the notion of being able to achieve complete consensus in decentralised systems. In large scale distributed systems, there are physical and logical limits that make a strong form of consensus for read/write variables impractical and ultimately impossible, and he claims existing solutions attempt to resolve these problems by assuming network latency is negligible and that independent agencies are reliable. In Khare's model of a decentralised system, agents store local models that estimate the actual value of the shared or global variable. The estimated value may be improved by an Assessor component that coordinates with trusted neighbouring agents who provide feedback on their estimates of the value.

When a group of agents share information about their estimates and converge on similar models, Khare describes the system as having the induced (or emergent) property of approximate agreement about their views of the system. In a system with approximate agreement between decentralised agents, their internal models should still be in agreement for a significant percentage of time, and approximate agreement can be used to coordinate the adaptive behaviour of the agents.

Summary

Khare presents a novel model of a decentralised architectural style that adds components to a REST system to handle sources of uncertainty, centralised, and decentralised control in distributed systems. In particular, he suggests replacing the consensus model for establishing the value of shared variables, found in existing consensus-based distributed control models, with estimated variables. The model has similarities with CRL, in that it makes use of caches to store local views of estimated variables and allows independent agents to share feedback information relating to estimated variables to help improve their estimations of the true value of the variable. However, the software architecture model is not dynamic and the model does not support the construction of self-adaptive or autonomic software.

2.5.6 Control Theoretic Approaches to Building Autonomic Systems

The possibility of modelling self-adaptive and autonomic systems as closed control systems has recently been investigated independently by Hinnelund (Hinnelund, 2004), Taylor (Taylor and Tofts, 2004) and Li (Li, 2000). Control engineering is a field of engineering concerned with optimising the performance of engineered artefacts with respect to measures such as energy usage, reliability and velocity of mechanical motion (Dutton et al., 1997). In control engineering, a controller typically attempts to maintain system operation within some given region in the face of externally supplied disturbances by reacting to monitored changes in the system and by anticipating future changes in the system. When designing control systems, engineers ensure that they avoid properties such as too slow convergence, oscillation, or chaotic behaviour (Taylor and Tofts, 2004).

In control systems, outputs of a system are measurable quantities to be controlled and inputs are manipulable. A control system can be either a closed-loop regulatory system (see figure 2.8) or an open-loop regulatory system that contains no feedback loop (Dutton et al., 1997). Taylor sees

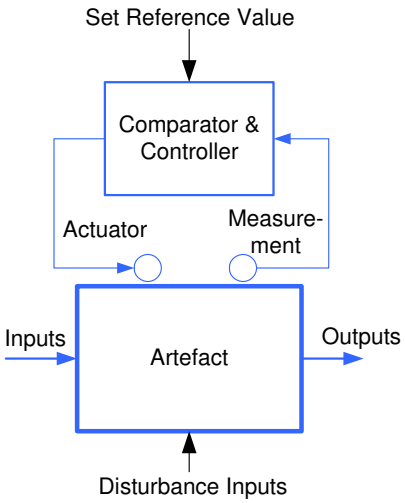


Figure 2.8: Closed-loop control systems.

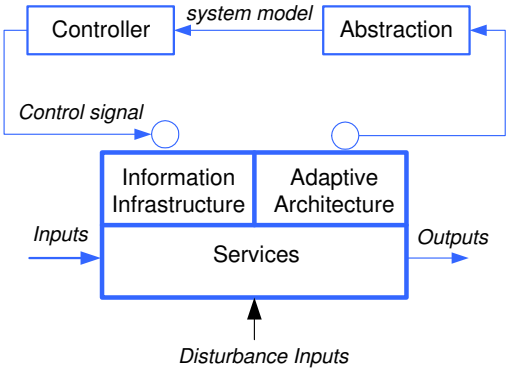


Figure 2.9: Hinnelund's model of an autonomic computing system as a control system.

autonomic systems as examples of closed-loop control systems (Taylor and Tofts, 2004). Inputs that cannot be controlled directly are called disturbance inputs. Relationships between inputs and outputs can be linear or non-linear, with linear systems being favoured over non-linear systems due to their predictability.

State Monitoring

A state-space model can be used to describe a control system that has more than one input and/or output. The size of the state-space should be tractable. The monitoring of system states is equivalent in control systems terms to soliciting feedback from system operation in order to guide the adaptive behaviour of the system (Dutton et al., 1997). State information in control systems usually satisfies the Markov property (Dutton et al., 1997), as the decision making components generally make decisions without recourse to historical information (Hinnelund, 2004; Dutton et al., 1997). Techniques have been developed, such as estimators and observers (Dutton et al., 1997), to handle situations where measurements of system state cannot be made directly and must be computed from available measurement data.

In Hinnelund's model (see figure 2.9) there is no support for an abstract model of system state. He provides a sample autonomic system in (Hinnelund, 2004) that implements a service workflow. In

the system, sensor data originates in sensors and arrives via state processing (compute) nodes at the command centre. The states available in this implementation consist of states for sensor mode and network mode at sensor nodes, states for the current configuration, the connection quality, and for the network stress at state processing nodes.

Adaptation Actions

In a control system, adaptation actions are performed by effecting actuators (see figure 2.8). Hinnelund does not provide a formal model for actuators or system adaptation actions. In his sample system presented, the adaptation actions reconfigure the connections in a workflow configuration.

Adaptation Consistency

In Hinnelund's Master's dissertation, he does not present a model for intrusive adaptation actions, so he does not explicitly treat the adaptation consistency problem. In his sample system, the adaptation actions are intrusive and system integrity is maintained using type-safe reconfiguration operations. He also meets the mutual consistency requirements by only adapting stateless component connections in the workflow and managing outstanding communication. The system does not have to deal with maintaining application state invariants.

Decision Policy

In control systems, the comparator and controller subsystems make up the decision making component of a control system. The relationship between the inputs and outputs is typically captured using mathematical models and manifested in the comparator and controller subsystems.

In Hinnelund's model, each node has a decision making component called a learner that encapsulates a 1^{st} party decision policy. The decision making component uses a feedback mechanism of rewards from a global system critic to find an optimal decision policy using machine learning. A learner's decision policy maps observed system states onto control outputs, while system knowledge is implicitly stored by the critic in the form of a reward model.

In Hinnelund's sample workflow system, there is homogeneity in the decision policy among learners in the system. The workflow system consists of sensors, state processing nodes and a command centre. The optimisation goals of the system are to maximise the amount of data delivered to the command centre and also minimise the age of the data. Each learner has a set of possible configuration actions (e.g., set sensor coverage area, set workflow routing to shortest delay), some system states to monitor, (concerned with acquiring sensor information), and a decision policy that is updated by receiving rewards for taking actions.

Evaluating and Updating the Decision Policy

In the model, a reward provides evaluative feedback on the success of an adaptation action in achieving the optimisation goals of the system and is used by the learner to improve the decision policy.

The critic's reward model is based on a linear model:

$$r = c.\alpha - (1 - \alpha)p$$

where r is the reward, the coverage c is the union of the areas reaching the command centre, discounted by the delay in data reaching it, while p is the total number of packets sent and α is a bias parameter towards maximising sensor coverage or minimising communication costs.

Hinnelund provides a model for learning adaptive behaviour based on an iterative policy search algorithm, a RL learning algorithm, see section 2.4.3. The algorithm is run over a series of episodes where each state-action pair (s, a) visited produces a reward by a centralised system critic after each episode. The value function used in his experiments for learning the decision policy is:

$$Q(s,a) = \frac{\sum_{i \in z_{s,a}} r(i)}{\|z_{s,a}\|}$$

It calculates the mean reward produced by the episodes with the state-action pair (s, a), where $r(i)$ is the reward received in episode i and $z_{s,a}$ is the set of episodes where (s, a) was visited. The best policy is determined using

$$\pi_N^b(s) \quad = \quad \max_a Q(s,a)$$

As Hinnelund notes, the rate of exploration must be low for this algorithm to converge and should be inversely proportional to the number of nodes in the system and the time units used.

Coordination

Although there is no explicit support for coordination between decision making components in Hinnelund's system, he noted that a correlation may exist between the decision policies on nodes. When a change of behaviour in one node affects the expected distribution of rewards over state-action pairs in other nodes, Hinnelund states that it "causes implicit collaboration to form among node types as the policy is being improved".

Summary

The use of control systems theory to build autonomic systems is a recent and immature area of research. Hinnelund presents a vision of what a model for building autonomic systems as control systems may look like, although he does not provide a complete system model for building autonomic applications. The iterative policy search algorithm, however, represents a first step towards building components that can learn autonomic behaviour. The system developed uses components that depend on a centralised system critic to supply rewards. As such, additional support is required for the decentralised coordination of both components and learners to enable the construction of decentralised systems that can learn autonomic behaviours.

2.5.7 Other Related Systems

The research areas of reflection, self-adaptive systems and dynamic software architectures have produced an extensive number of systems that have either influenced the design of the K-Component model or have features in common with K-Components.

Fractal is a self-adaptive component model (David and Ledoux, 2003) that separates out a component's adaptation concerns into a meta-level adaptation policy. A component's adaptation logic is specified using the event-condition-action paradigm and meta-level programs reason about adaptation conditions by registering interest in and monitoring context events that provide feedback regarding the state of components and connectors. Context events can be generated locally or remotely by a context awareness service and are used as a trigger for self-adaptive behaviour. Fractal is implemented in Java and is based on the more traditional synchronously scheduled reflection programs that operate on MOPs.

Marmol is a meta-level framework that uses architectural reflection to build dynamic architectures that express their dynamism at the meta-level (Cuesta et al., 2002b,a). It provides an ADL, called PiLar, that can be used to explicitly specify a consensus-based dynamic coordination architecture. Pilar consists of both a declarative structural language and an imperative dynamic language that defines adaptation logic for the architecture.

Kon's automatic dependency management model for components has influenced the design of component dependency management in the K-Component model (Kon, 2000). His component configurator manages incoming and outgoing dependencies between components and enables the asynchronous delivery of events between components. He does not provide a specification language, however, for components. The Rapide component model, however, does contain a component specification language, where components specify their service provision and service requirements using *provides* and *requires* interfaces respectively. These interfaces are used for synchronous communication, and *in* and *out* actions are available for the communication of asynchronous events between components (Luckham and Vera, 1995; Luckham, 1996). Rapide, however, has no support for connectors as first-class entities and does not provide a mapping to standard programming languages such as Java or C++ (Medvidovic and Taylor, 2000).

The K-Component framework is built on top of CORBA middleware and has to handle adaptation consistency problems when reconfiguring CORBA applications. There has been existing work on dynamically reconfigurable CORBA systems by Almeida (Almeida, 2001) and Sadjadi (Sadjadi and McKinley, 2004) that address adaptation consistency issues when reconfiguring CORBA-based applications.

K-Component's model of asynchronous reflection is conceptually closest to Brazier's model of reflective agents that reason about their own behaviour and other agents' behaviour (Brazier and Treur, 1999). She states that reflective agents must be able to perform reasoning (Brazier and Treur, 2000) about:

- the external world
- the agent's interaction with the external world

Reviewed System	Techniques
QuO	Aspect-Oriented Programming (AOP)
OpenORB	Arch. Reflection, O-O Reflection, Component Frameworks
Accord	Not Specified
Georgiadis	Dynamic Software Architecture
Khare	None
Hinnelund	Iterative Policy Search using a Centralised Critic

Table 2.1: Comparison of techniques used by reviewed systems.

- the agent's internal processes
- other agents' internal processes
- interaction between agents
- agent specific tasks

However, Brazier's model is not aimed at building self-adaptive or autonomic software, but at describing how reflection provides a model for autonomous agents reasoning about their goals and constructing plans in a social, multi-agent context. She does not present a programming model for asynchronous reflection.

Architectural approaches to self-healing and autonomic computing have been followed by Garlan (Garlan and Schmerl, 2002), and White et al. (White et al., 2004) respectively. Both of these approaches are based on centralised management services to support the adaptation of system components and are not suitable for decentralised environments. However, White's model includes the interesting notion of a utility function decision policy that can be used to compute the optimal action to take given the system state. In (Chess et al., 2004), a utility function decision policy is used by a centralised resource arbiter to optimise resource allocation among nodes in a closed distributed system. The approach has similar system optimisation goals to CRL, but the utility function is a centralised optimisation technique.

IBM's autonomic computing model is based on the notion of autonomic elements that each contain exactly one autonomic manager (Kephart and Chess, 2003; IBM, 2004). An autonomic element could be a resource like a database, server or software application. The model, however, is aimed at building autonomic distributed systems in closed environments that reside in a single domain of administration. The model uses a model of centralised control, using services such as resource arbiters, policy repository and resource managers, to guide the autonomic behaviour of managed computing elements.

2.6 Feature-Based Comparison of the Reviewed Systems

This section provides a feature-based comparison of the different models presented in this chapter with respect to various features, including the six requirements for a self-adaptive autonomic component model in section 2.3.

The first set of features that are compared are the techniques used to support self-adaptive behaviour by the different models (see table 2.1) . The support for monitoring system state, the type of adaptation actions supported by the system, and the adaptation consistency of those actions are often

	State Model	Adaptation Actions Supported	Adaptation Consistency
QuO	Middleware, System State	Intrusive, Non-Intrusive, Internal	√
OpenORB	AMM, Middleware, System State	Intrusive, Non-Intrusive, Internal	√
Accord	Component State	Intrusive, Non-Intrusive, Internal, External	Not Specified
Georgiadis	Software Architecture State, Component State	Intrusive, Non-Intrusive, Internal	√
Khare	Application State using Estimator Functions	Non-Intrusive	N/A
Hinnelund	Component State	Non-Intrusive, Internal, External	Not Specified

Table 2.2: Comparison of state models and adaptation actions.

	Decision Policy (DP)	Decl-Prog	DP Exec.	DP Evaluation	DP Updates
QuO	Action Policy, Rules, ECA	√	Async/Sync	Sys-Admin, GUI	X
OpenORB	Action Policy, Rules, ECA	X	Sync	Sys-Admin	Manager Comps
Accord	Action Policy, Rules	√	Async	Sys-Admin	Composition Agt
Georgiadis	Action Policy, Arch-Constraints	X	Async	Sys-Admin	Selector Fns
Khare	Estimator Function	X	Async/Sync	X	X
Hinnelund	Learning Policy with Critic	X	Sync	Critic/Learner	Iterative Policy Search

Table 2.3: Comparison of decision policies.

a product of the technique(s) used to build the system (see table 2.2). The state models refer to the set of subsystems from which each system can use state information in making adaptation decisions. Systems that support intrusive adaptation actions provide better support for building autonomic applications as they can adapt their structure to a changing environment. Intrusive adaptation actions should maintain adaptation consistency.

The second set of features (see table 2.3) covers the type of decision policy supported, whether the decision policy is scheduled to execute synchronously or asynchronously and how the decision policy can be evaluated and updated. Hinnelund's model is the only system that evaluates and learns decision policy autonomously. This feature is very useful when building large, autonomic systems where the set of available adaptation actions and system states may be too large for the specification of decision policies as action policies.

The final set of features compared (see table 2.4) are the types of system support for the coordination of components supported by the different systems. Khare's coordination model is the only technique that is suitable as a starting point for the construction of decentralised autonomic systems. It provides a model that describes how agents can induce agreement on the value of some unknowable variable through local views of the system and estimator functions. This is called emergent consensus as agreement between agents on their partial views of the system emerges from their interaction.

	Centralised	Distributed Consensus	Emergent Consensus
QuO	√	X	X
OpenORB	Limited	X	X
Accord	√	X	X
Georgiadis	X	Local Arch. Constraints	X
Khare	X	X	Decentralised Arch. Style
Hinnelund	X	X	X

Table 2.4: Comparison of coordination models.

2.7 Summary

This chapter reviewed a number of models that support the construction of self-adaptive and autonomic computing systems. We also presented models for learning autonomic behaviour and for the decentralised coordination of components. Most of the self-adaptive systems reviewed provide support for the specification of self-adaptive behaviour as action policies and those that allow intrusive adaptation actions provide infrastructural support for maintaining adaptation consistency. Hinnelund's model is of interest as it addresses the problem of learning a decision policy using a reward model for adaptation actions. Decision policies that encapsulate a system's adaptation logic are executed either synchronously or asynchronously, with no existing reflective system providing support for specifying general adaptation logic that is executed asynchronously. Khare's model addresses the problem of coordinating agents in decentralised systems by providing a model for estimating the value of shared or global variables, although the model is not designed for building self-adaptive or autonomic systems.

None of the existing architectures addresses all the challenge problems of the learning of self-adaptive behaviour and the coordination of self-adaptive components to establish system-wide autonomic properties in dynamic, uncertain environments. These issues are addressed by the K-Component Model presented in the following two chapters.

Chapter 3

The K-Component Model

> "Reflection ... slackens the intentional threads which attach us to the world and thus brings them to our notice; it alone is consciousness of the world [because it reveals that world as strange and paradoxical.]"
>
> Maurice Merleau-Ponty, Phenomenology of Perception (1945)

This chapter describes the K-Component model that addresses four of the six requirements for a self-adaptive, autonomic component model identified in section 2.3. These include the provision of a state model to describe the operating state of a system, adaptation actions to adapt the system at runtime, a decision making component that uses a decision policy to determine the adaptive behaviour of the system and a mechanism to automatically evaluate and update the decision policy over time. The maintenance of system integrity and consistency during adaptation is dealt with in the implementation of the K-Component model in chapter 5 and the requirement for a decentralised coordination model is addressed by the collaborative reinforcement learning algorithm in the next chapter.

The K-Component model is described in this chapter as a programming model and architecture for building autonomic systems based on self-adaptive component software. The features of the programming model include component specification in K-IDL, a component description language that extends CORBA's IDL (OMG, Dec. 2002), and adaptation logic specification in the adaptation contract description language (ACDL). The architecture consists of a set of management components and a framework used to generate components, connectors and adaptation contracts from K-IDL and ACDL. The main abstractions of the K-Component model and the motivation behind the design decisions are discussed, independent of its implementation as a distributed computing platform. In the review of existing self-adaptive and autonomic software models in the previous chapter, the shortcomings of existing architectures in building autonomic distributed systems were highlighted. The model presented in both of the following chapters addresses these shortcomings.

3.1 Introduction

The K-Component model provides a programming model and architecture that enables the construction of a self-adaptive system that can monitor the state of its operation and environment, and perform

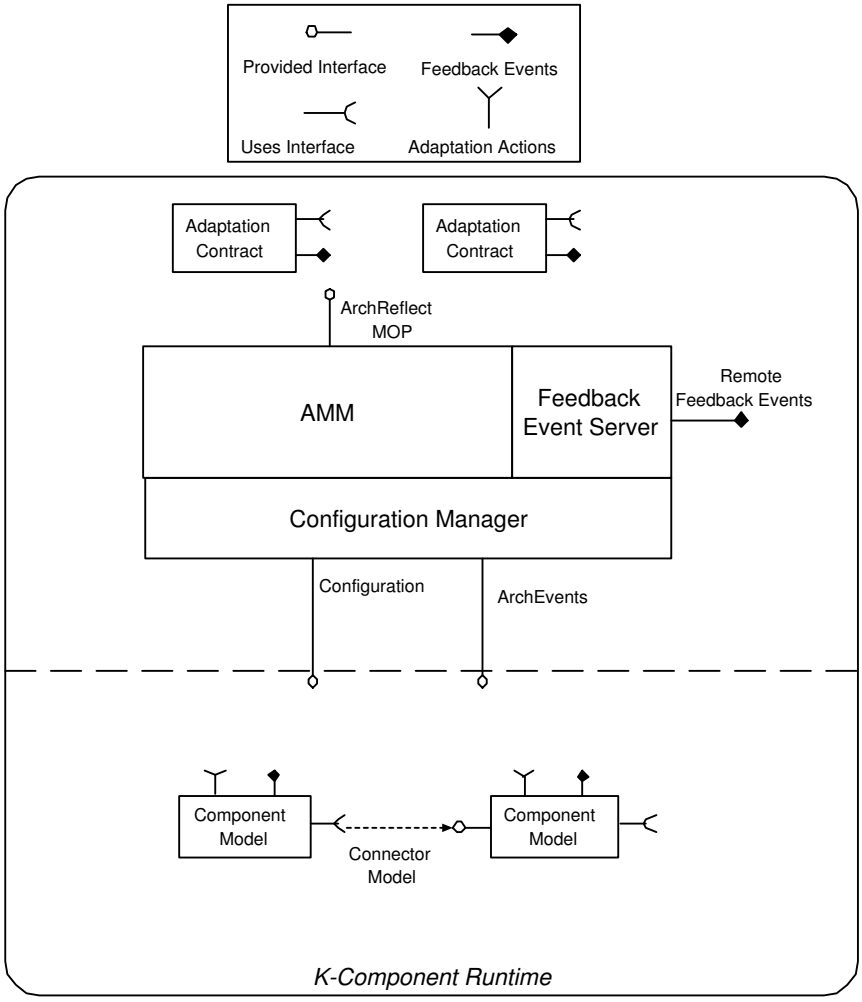


Figure 3.1: A K-Component.

conditional adaptation of its structure or behaviour, e.g., to self-heal or self-optimise the system. A K-Component is a runtime with a single address space (see figure 3.1) where components reside and dependencies between components, whether internal or external to the runtime, are managed using connectors. The eight basic concepts that make up the K-Component model are:

1. components
2. connectors
3. the architecture meta-model (AMM)
4. feedback states
5. adaptation actions
6. feedback events
7. adaptation contracts
8. asynchronous reflection

In addition to these concepts, a configuration manager is deployed in each K-Component runtime to provide thread management and configuration services.

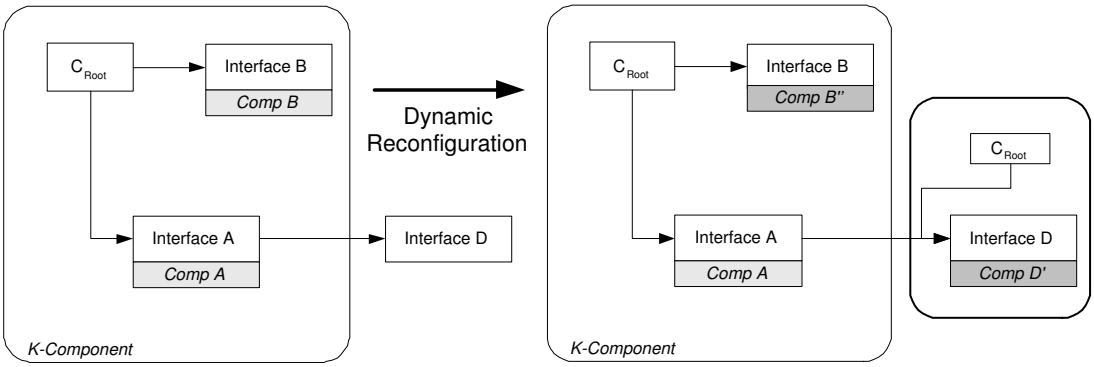


Figure 3.2: Dynamic software architecture reconfiguration.

Architectural reflection is used to reify the structure of components and connectors in a K-Component runtime as an AMM. An AMM captures the dynamic representation of the K-Component as a configuration graph, with interface/component pairs as labelled nodes and connectors as edges (see figure 3.2). Components are provided as a unit of composition that encapsulate their implementation behind a strongly-typed interface and specify all their explicit dependencies as required interfaces. Connectors are runtime objects that mediate communication between components and provide reconfiguration functionality. Feedback states provide runtime information about the operating status of components, i.e., states defined on component interfaces, and connectors. Feedback events are defined as predicates on feedback states and support the asynchronous communication of changes in feedback state between K-Components. Adaptation actions are provided as both architectural adaptation actions that reconfigure the AMM and component-specific adaptation actions.

Autonomous reflective programs, called adaptation contracts, operate on the AMM and reason about adaptation conditions using feedback regarding the state of components and connectors. Adaptation contracts implement a 1^{st} party decision policy for a component in the K-Component runtime. Its adaptation logic is specified declaratively in the adaptation contract description language (ACDL). Adaptation logic in adaptation contracts can be specified as an action policy, using `if-then` rules or the ECA paradigm, or as a learning policy using RL or CRL. An asynchronous implementation of computational reflection interleaves the execution of adaptation contracts with system operation, enabling adaptation contracts to continuously monitor and affect system operation.

3.2 Objectives

The goal of the K-Component model is two-fold: to provide a self-adaptive component model and support the construction of coordination models for building autonomic distributed systems in decentralised environments. Existing models for building autonomic distributed systems assume the availability of system-wide information to guide the autonomic behaviour of components and are often designed to operate in closed environments (White et al., 2004; Liu and Parashar, 2004; Nowicki et al., 2004).

In considering the inclusion of support for self-adaptation in a software system, the requirements from section 2.4.1 need to be addressed by the K-Component model in order to be able to build distributed autonomic systems. These include monitoring system states to gather information about

system operation, providing adaptation actions that maintain system integrity, providing a 1st party decision policy that can itself be evaluated and updated at run-time and supporting a decentralised coordination model to establish and maintain system-wide properties. In addition to meeting these requirements, the K-Component model builds on previous work in building self-adaptive software using compiled reflective programming languages (Dowling et al., 2000) and addresses the limitation of reflective code being executed synchronously with program execution.

A further objective of the K-Component model is to provide a programming model that simplifies the construction of self-adaptive software by a programmer. The specification of explicit dependencies between components enables the automated generation of the AMM and the provision of a declarative programming language allows programmers to express adaptation logic at a high level, enabling the clearer expression of their intentions.

The evaluation of K-Components as a model for building autonomic distributed systems is performed in chapter 6. A set of experiments investigate the autonomic capabilities of the system built using K-Components to evaluate if it is able to:

- self-heal a component in the case of a failure of one of its connections
- self-optimise a component by adapting its internal behaviour in response to changes in its feedback state or a feedback state in a component or connector it uses
- self-optimise system performance in a changing, decentralised environment using CRL.

3.3 Asynchronous Reflection

Asynchronous reflection is a model of computational reflection where autonomous reflective programs reason about and act upon a system asynchronously to the operation of the system. Asynchronous reflection is a goal-directed process that is not separable from the system on which it is operating. Its essential characteristics are observation and outputs (new knowledge, system adaptation, plans of action, etc). Ideally, it should be able to acquire new knowledge, store that new knowledge and in turn reflect on that new knowledge, leading to towers of reflection.

Synchronous Reflective Systems

Computational reflection has been used to design open and extensible reflective programming languages as well as open and extensible meta-level architectures. Reflective object-oriented programming languages were designed to provide open implementations of the programming languages so that programmers could modify the languages' object model (Schaefer, 2001), for example to support the transparent addition of fault-tolerance (Killijian and Fabre, 2000), concurrency (Haraszti et al., 2001) and persistence (Haraszti et al., 2001) to programs.

In existing compiled, reflective object-oriented programming languages, such as Iguana v2 (Schaefer, 2001) and OpenC++ v2 (Chiba, 1995), the execution of reflective code is tightly-coupled with program execution. Reflective code is executed synchronously with program execution at reification (intercession) points in the programming language's object model. This adds an often substantial fixed

overhead to request processing at the various reification points in the object model where reflective code is executed (Dowling et al., 2000).

Synchronous reflective programming languages generally realise the causal connection between the base level and meta-level as an implementation link. Reflective code can only be inserted/removed at the reification points and is generally executed by an application thread, i.e., it is executed synchronously with program execution.

Existing compiled, reflective object-oriented programming languages also do not reify, or make available for monitoring, state information concerning entities external to the application, even though this state information is useful when monitoring an application's dependencies or environment for adaptation conditions. They also support the unconstrained adaptation of applications and do not provide the reconfiguration management infrastructure necessary to maintain application consistency during reconfiguration.

However, when considering reflection for building self-adaptive software, designers must also handle conditions that require the adaptation of software arising due to changes in the application's dependencies or environment. As the occurrence of external adaptation conditions is often orthogonal to system operation in autonomous components in a distributed application, an application that supports computational reflection for self-adaptation should be able to identify adaptation conditions in its environment independent of normal program execution. A computer system that performs computational reflection for adaptation should reflect asynchronously on its operation.

3.3.1 Asynchronous Reflection for Self-Adaptive Software

Reflection for adaptation has more similarities with the theory of intelligent agents (Wooldridge and Jennings, 1995) than the more traditional use of synchronous reflection for building open and extensible languages or meta-level architectures. Intelligent agents are characterised by autonomy, social ability, reactivity and proactiveness (Wooldridge and Jennings, 1995). Autonomy and proactiveness allow a decision making agent to reason about a system's operation and adapt the system's operation to meet some programmer supplied goal. K-Component's model of asynchronous reflection is closest to Brazier's model of agents that perform reflective reasoning about their own behaviour and other agents' behaviour (Brazier and Treur, 1999, 2000).

A system that supports asynchronous reflection requires at least one autonomous reflective program dedicated to reasoning about the system and acting upon the system. In a self-adaptive system built using asynchronous reflection, the reflective program can be a decision making component that monitors the system for adaptation conditions and performs adaptation actions. Asynchronous reflective computation adds overhead to application processing time, but it is a programmer-defined amount of overhead as reflective program are configured to be executed periodically at a *sampling time interval*, t_c. In self-adaptive systems, a trade-off can be found between the extra overhead of reflective computation and the responsiveness of the system to adaptation conditions.

3.3.2 Reification Categories for Self-Adaptive Software

The aspects of a system or language's internal representation that are made explicit in meta models are called the reification categories of the system. Reflective computation for adaptation requires the reification of semantic information about potential adaptation conditions in the system, so a reflective component can monitor the system for adaptation conditions. In K-Components, the AMM is reified, allowing the observation and manipulation of the system as a configuration graph of components and connectors. However, additional semantic information about a system is also reified for observation in the AMM:

- feedback states defined on components and connectors
- feedback events defined as predicates on component feedback states

Feedback states are designed to provide information about the operating status of a component or connector. Feedback state values can be observed, but not modified, by reflective programs. This is because they represent concrete metrics describing component or connector operation. Reflection on the feedback states of components is performed by reflective programs. Feedback events, however, are designed to provide asynchronous notification about changes in the value of a component's feedback state, when its predicate is matched.

3.3.3 Weakening Consensus between Meta Models and the Base-Level

Asynchronous reflection can loosen the requirement for full consistency between meta models and the base-level for particular reification categories. In K-Components, the causal connection between base-level and meta models is a combination of an intercession link from base-level code and a representation link from the meta models. Similar to synchronous reflective systems, reification of architectural events, including component creation/deletion, connector creation/deletion and connector binding/unbinding, from the base-level to the AMM is performed at intercession points in component and connector objects. However, the reification of feedback state values for components and connectors is performed by a thread that periodically synchronises the value of feedback states of components and connectors in the runtime with cached values. The feedback event manager also receives feedback events from remote components and uses them to update the values of the cached feedback states. The caching of feedback event values is useful for systems operating in decentralised environments, as it provides them with a local, estimated model of the operating status of remote components.

Reflection on feedback states is implemented as a representation link. Reflective programs can query feedback state values using an interface, called the ArchReflect MOP, but there is no guarantee of consensus between AMM and component or connector values for the feedback states. The AMM caches the last observed value of a component's feedback state and queries may return stale values for the component feedback state. This lesser form of consensus, between the AMM representation of feedback state values and their actual values, only guarantees that the cached values in the AMM will never be more out-of-date than either the sampling time interval or the elapsed time since the last received feedback event. The CRL algorithm, introduced in chapter 4, operates on these cached,

estimated feedback state values and also provides a collaborative feedback model to help improve the accuracy of the cache. This is further discussed in chapter 4.

3.4 Component Model

A software component is defined by Szyperski as:

> "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parities (Szyperski, 1998)."

Components in the K-Component model specify the services they provide and the services they require as interfaces. The support for "provides" and "uses" (or required) interfaces allows the definition of dependencies between components (Luckham and Vera, 1995; Dowling and Cahill, 2001b; Kalibera and Tuma, 2002). Kon showed in (Kon, 2000), how the explicit management of inter-component dependencies is necessary to maintain adaptation consistency in dynamically reconfigurable component-based systems. A component specification language, K-IDL[1], is used to describe a component interface as:

```
component <name> {
  provides <Interface>;
  [uses <Interface> <uname>;]*
  [state <sname>;]*
  [action <aname>;]*
};
```

A component defines a single provided interface, `provides <Interface>`, to describe the functionality it offers. It also defines zero or more uses interfaces, `uses <Interface> <uname>`, that represent the interfaces required by the component to meet its contractual obligations. Uses interfaces make explicit the dependencies of the component on other components and `<uname>` maps to a connector object defined as a member variable in the component. A component can also define zero or more component feedback states, `state <sname>`, that are implemented by component providers and should provide feedback on the operating state of the component at runtime. Finally, components can define zero or more actions, `action <aname>`, that represent adaptation actions and are implemented by the component provider. Actions can be invoked by reflective programs at runtime to adapt the component's operation.

Components only provide a single interface. There is no support for composite components that provide more than one interface. This support is not provided in the model, since such a composite component can be built in K-Components as a configuration of connected components. Both `provides` and `uses` interfaces of components are compiled into connector objects that are encapsulated in a component instance. The connectors mediate all communication between components, and there is

[1] K-IDL syntax was influenced by IDL-3 in the CORBA Component model (OMG, 1999).

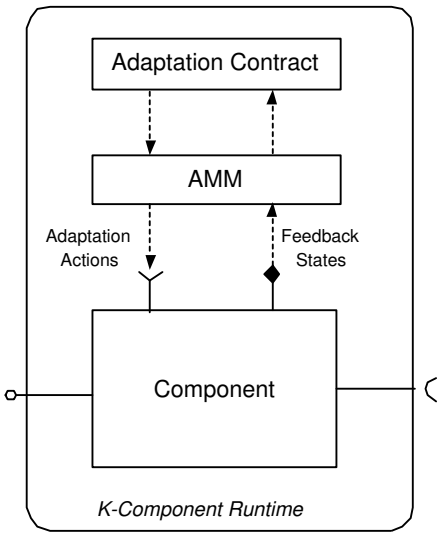


Figure 3.3: The component model and an adaptation contract.

no support for shared memory between components. Each component is responsible for managing its own state.

Component feedback states are defined in the component interface using K-IDL but they are not part of the public interface of the component and are only observable by external K-Components using feedback events. Component feedback state simplifies the modelling of a decision policy for adapting components, as feedback state describes the operating state of a component at a particular point in time. Feedback state that describes time dependent properties of a component is particularly useful. For example, feedback states can be used to model components as dynamical systems (De Wolf and Holvoet, 2003) and feedback states can be used to provide performance metrics for a component. These performance metrics can be used to evaluate adaptation actions that were performed on a component. Component feedback state is represented as a scalar. This provides a uniform state model for describing component operation that helps the construction of a decision policy that can reason about component operation.

Adaptation actions are also defined on the component in K-IDL, but again they are not part of the public interface of the component and can only be invoked by adaptation contracts using the ArchReflect MOP. Component adaptation actions are implemented by the component provider, typically as fine-grained adaptation actions that only adapt the component on which they are defined, e.g., to change the implementation strategy for an internal algorithm to improve component performance (Kiczales et al., 1997; Dowling et al., 2000). Adaptation actions provide a declarative interface that accepts a strategy as a parameter, indicating some strategy for executing the adaptation action. Adaptation actions can also supply evaluative feedback in the form of a scalar reward to the adaptation contract to indicate the success or otherwise of the adaptation action. Evaluative feedback can be used to evaluate and improve the performance of a decision policy for the component.

3.4.1 Reflective Component Model

In the K-Component model, components are units of computation and do not reflect on their own operation. Adaptation contracts are provided as autonomous, reflective programs that reflect asyn-

chronously on the operation of a component (see figure 3.3). The goal of an adaptation contract is to optimise the component's operation by adapting the component's structure and behaviour to a changing environment.

Components can be reflected upon by adaptation contracts as they reify aspects of their representation in the AMM. A component's provided and uses interfaces are reified in the AMM as nodes and edges in the configuration graph, respectively. A component's feedback states and adaptation actions are also reified for observation and execution, respectively. The value of a component's feedback states can be monitored and its adaptation actions can be executed at runtime using the ArchReflect MOP (see section 3.6.4). Component instances in the AMM can also be introspected, inserted, removed and replaced at run-time.

3.4.2 Definition of a Component

A definition of a component is provided for completeness. Later definitions of connectors, the AMM and adaptation contracts are also provided. These definitions are required, as they are referenced by definitions of decision policies in this chapter and the next chapter.

Definition 3.1: A component interface is $i_i = \{interface, U_i, F_i, A_i\}$, $i_i \in I_k \in \mathcal{I}$, where

- i_i is a component interface, I_k is the set of component interfaces in the runtime k and $\mathcal{I}$ is the set of all component interfaces in all runtimes in the system.
- $interface$ is strongly typed and defines the provided services.
- $U_i = \emptyset \vee \{interface_1, ..., interface_M\}$, a finite set of zero or more required interfaces.
- $F_i = \emptyset \vee \{f_1, .., f_N\}$, a finite set of set of zero or more component feedback states.
- $A_i = \emptyset \vee \{a_1, .., a_O\}$, a finite set of zero or more component adaptation actions.

Definition 3.2: A component instance, or simply a component, is $c_i = \{i_i, id, L_{c_i}\}$, where

- c_i is the component and $c_i \in C_k$, C_k is the set of components in the runtime k and $C_k \in \mathcal{C}$, where $\mathcal{C}$ is the set of all components in the system.
- id is the identifier for the component in runtime k.
- $L_{c_i} \in L_k$, the set of outgoing connectors (see section 3.5.2) encapsulated in c_i.

Reflective Operations on Components

The operations that can be executed by clients of a component are defined in the component interface. There are also a set of mostly introspection functions defined on every component that are used to reflect on component operation:

- $getProvides(c_i) \rightarrow \{interface\}$, gets the provided interface for c_i.
- $getConnector(c_i \times id) \rightarrow \emptyset \vee l_i \in L_{c_i}$, gets the connector identified by id deployed in c_i.
- $getConnector(c_i \times interface) \rightarrow \emptyset \vee \{l_0, .., l_i\} \subseteq L_{c_i}$, gets all the connector objects deployed in c_i that implement $interface$.

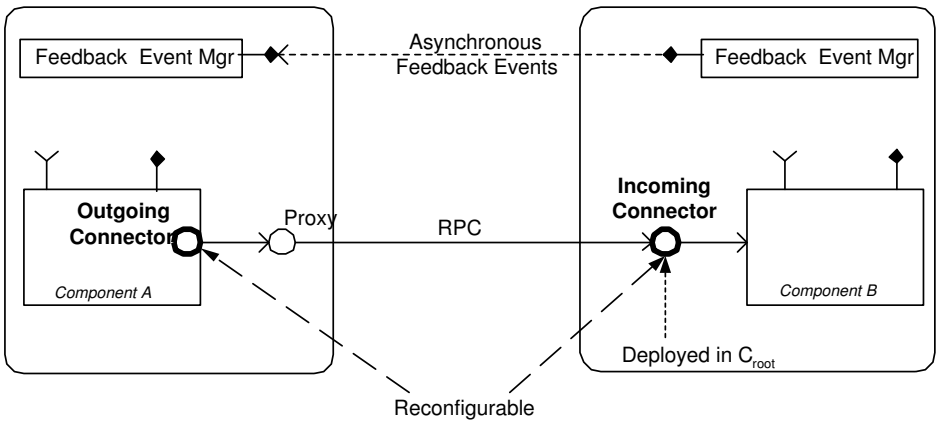


Figure 3.4: Connector style for decentralised environments.

- $feedback : (c_i \times f_j) \rightarrow \mathbb{R}$, gets the value of feedback state f_j on c_i as a scalar.
- $ID : c_i \rightarrow id$, returns the unique identifier for c_i.s
- $action(c_i \times a_j \times strategy) \rightarrow \{r\}$, $strategy, r \in \mathbb{R}$, executes adaptation action a_j on c_i with an optional strategy parameter represented as a scalar and returns a scalar reward, r, indicating the success of the adaptation action.
- $passive : c_i \rightarrow true \vee false$, returns true if c_i has no ongoing computation and has no ongoing communication with other components, i.e., it is passive (Wermelinger, 2000).

3.5 Connector Model

The connector abstraction is important in distributed systems as it abstracts the communication protocol and encapsulates the interaction protocol between communicating components. According to Fielding (Fielding, 2001):

> "A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components."

Connectors support the explicit, dynamic binding of one component to another. Connectors often also support the dynamic unbinding of a connection to a component. Reconfigurable connectors are connectors that support binding and unbinding operations, and have been used as a mechanism to build adaptive software, such as dynamic software architectures (Wermelinger, 2000; Moazami-Goudarzi, 1999). Reconfigurable connectors allow the adaptation of the connections between components at runtime. Adapting connections between components is often preferable to adapting the behaviour of the components to one another as it allows us to treat components as pluggable black-box entities.

3.5.1 Connectors for Decentralised Environments

A model of connectors for use in decentralised environments has to address a number of issues not encountered by existing models for reconfigurable connectors in a client-server environment (Wermelinger, 2000; Moazami-Goudarzi, 1999). A well-known problem with the client-server paradigm

in decentralised environments is that the interaction protocol provided by standard RPC connectors assumes tight coupling between components that are subject to frequent changes in availability due to network dynamism (Khare, 2003). Ideally, RPC connectors should be able to identify and adapt to conditions such as component unavailability, suboptimal connection quality and any other factor that can affect optimal system operation. However, they are not able to identify adaptation conditions that are inherently asynchronous events, such as failures (e.g., component or connection failure may occur while connectors are idle) and component feedback state changes (that may require adaptation actions).

Another feature of the client-server paradigm is that reconfigurable systems were often motivated by the need to change the implementation strategy of a server (Kiczales et al., 1997; Zinky et al., 1997; Almeida, 2001; Sadjadi and McKinley, 2004), e.g., to improve performance for its clients. In these systems, servers (or a 3rd party decision making server) often taken reconfiguration decisions on behalf of clients and can even initiate client reconfiguration (Almeida, 2001). However, in decentralised, autonomic systems we can expect that reconfigurations may be motivated by other conditions, such as the availability of better versions of services in the client's locality or the change of availability of a server due to network dynamism. In decentralised environments, there is more information available to clients to determine its optimal service than to centralised servers that have only as partial view of the system. As a result, in a decentralised environment, the responsibility for determining adaptation conditions and performing reconfigurations is shifted from the server-side to the client-side, i.e., the agents.

A different style of connector is required for decentralised environments that addresses these problems. In K-Components, the connector model provides reconfigurable RPC-style connectors for inter-component communication and feedback events to communicate asynchronous adaptation conditions between components (see figure 3.4). The asynchronous event style complements the RPC-based interaction protocol by providing the implicit signalling of adaptation conditions between connected components.

The RPC connector consists of a reconfigurable client-side *outgoing connector* and a reconfigurable server-side *incoming connector*. Reconfigurable outgoing connectors can support self-healing and self-optimisation by enabling clients to rebind from unavailable or suboptimally functioning components to alternative component implementations. Support is provided in K-Components for outgoing connectors to change their communication protocol at runtime by dynamically loading stubs for remote services. Incoming connectors are also reconfigurable and can be rebound to a different component implementation at runtime. The reconfiguration operations are designed to meet the structural integrity and mutual consistency requirements from section 2.3.3, by implementing a reconfiguration protocol that provides RPC-consistency (Almeida, 2001) between connectors and components during reconfiguration. RPC-consistency ensures that before a connector is reconfigured, all communication through the connector has terminated or is blocked before the reconfiguration operation is initiated.

3.5.2 Definition of a Connector

In K-Components, connectors are runtime objects generated from provides and uses interfaces in a component definition. Outgoing connectors are reified and represented in the AMM as edges in the configuration graph, whereas incoming connectors are implicitly represented in the component interface and are represented in the AMM as a node in the configuration graph.

Connectors are encapsulated inside components, and the main difference between outgoing and incoming connector objects is that incoming connectors are encapsulated in the c_{root} component, i.e., a component that represents the runtime, whereas outgoing connectors can be encapsulated in any component (see figure 3.4). Incoming connectors are deployed in c_{root} as they represent the provides interface to an independently deployable and repluggable component, and there is no support for nested or composite components. Both types of connector provide a fixed set of feedback states concerning information about connection operation status and performance.

Definition 3.3: a connector is $l_i = \{io, c_i, w_j, id, F_{l_i}, stub\}, l_i \in L_k \in \mathcal{L}$, where

- l_i is a connector, L_k is the set of connectors in runtime k and $\mathcal{L}$ is the set of all connectors in the system.
- $io \in \{incoming, outgoing\}$, the connector is either an incoming connector or an outgoing connector.
- if $(io = outgoing)$ then $c_i \in C_K \wedge c_i \notin w_j$, else $c_i = c_{root_k}$, where $c_i \in C_K \wedge c_i \notin w_j$, c_i is the component in which l_i is deployed and is not the same as the target component.
- $w_j =< i_j, c_j >, i_j \in I_k \wedge c_j = \emptyset \vee c_j \in \mathcal{C}$, w_j is a component interface/component pair that represents the target interface and component instance of the connector. The component c_j may or may not be resident in the connector's runtime k.
- id is the identifier for the connector in runtime k.
- $stub \in \{LIB, .., IIOP\}$, is the current transport used by connector l_i.
- $F_{l_i} = \{status, f_1, .., f_N\}$, where $status$ is a feedback state indicating the status of the connection to the target component and F_{l_i} is the set of other connector feedback states defined on l_i. In the current implementation, only the $status$ feedback state is supported.

Reflective Operations on Connectors

A set of introspection and reconfiguration functions are defined on connectors that can be used to reflect on, and adapt, connector operation:

- $getTarget(l_i) \rightarrow \emptyset \vee c_j$, the current component to which the connector is bound or NULL if the connector isn't bound.
- $getSource(l_i) \rightarrow c_i$, the current component in which the connector is deployed.
- $getInterface(l_i) \rightarrow interface$, the public interface provided by the connector.
- $ID(l_i) \rightarrow \emptyset \vee id$, returns the unique identifier for l.

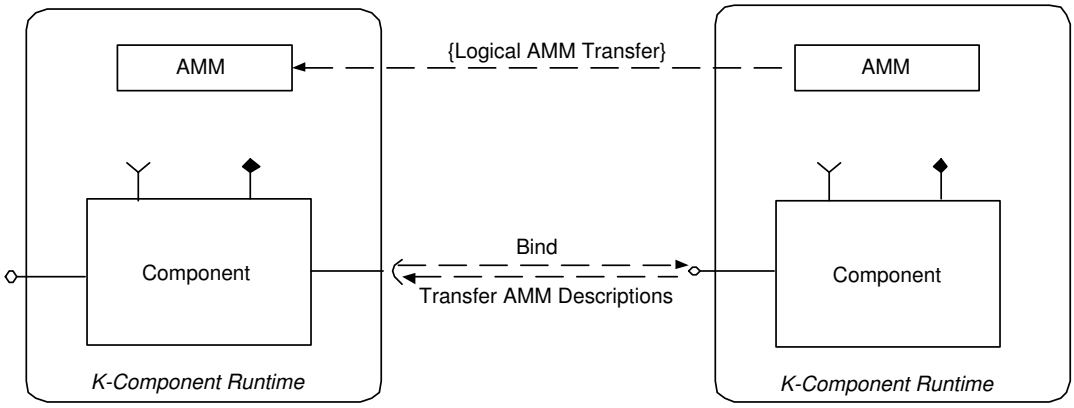


Figure 3.5: Abstract model of interaction between component binding and AMM transfer.

- $feedback(l_i \times f_j) \rightarrow \mathbb{R}$, gets the value of feedback state f_j on l_i as a scalar.
- $bind(l_i, c_j) \rightarrow l_i$, binds the connector l_i the component c_j if $getInterface(l_i) = getProvides(c_j)$.
- $unbind(l_i) \rightarrow l_i,\ where\ source(l_j) = c_i$

3.6 Architectural Reflection

Architectural reflection is concerned with the observation and manipulation of the configuration graph of a software architecture and its constituent nodes and edges (Dowling and Cahill, 2001b). A system that supports architectural reflection reifies its architectural features as an AMM that can be inspected and modified at run-time. Modifications of the AMM result in modifications of the software architecture itself, and the architecture is therefore reflective.

In K-Components, the AMM is modelled as a typed, directed configuration graph of components and connectors. A MOP called *ArchReflect* can be used by reflective programs to monitor and adapt the AMM at runtime.

3.6.1 Architectural Reflection and Dynamic Software Architectures

AMMs and dynamic software architectures share similar features and goals. They both aim to support the construction of dynamically adaptable software and they both provide an explicit representation of the system's configuration that is constructed dynamically, as client components discover and bind to services provided by other components, and can be safely manipulated at runtime. One main difference is that dynamic software architectures require the explicit definition of their software architecture using an ADL (Medvidovic and Taylor, 2000). Architectural reflection, however, automatically generates the AMM from both connection information in component descriptions and architectural events, such as connectors binding and unbinding components to and from one another (see figure 3.5). As such Medvidovic classifies architectural reflection as an implicit configuration language (Medvidovic and Taylor, 2000) and not an ADL. Architectural reflection, however, also causally connects a K-Component's AMM to the components and connectors in its runtime, so that changes in the AMM result in changes in the underlying components and connectors, making the system reflective. A registry of active components and connectors, called the KOM registry, provides the causal connection

from the AMM to the actual components and connectors (see figure 3.6). Components represented in an AMM that are external to its runtime cannot be explicitly adapted though, as this would require support for external adaptation actions, introducing access control and authorisation issues that are not addressed in this thesis.

3.6.2 A Self-Adaptive Architectural Style for Decentralised Systems

The K-Component AMM is designed to enable the construction of self-adaptive software in decentralised systems. In contrast to existing models of distributed systems that manage global software architectures for distributed systems using consensus-based techniques (Allen, 1997; Medvidovic et al., 2000; Georgiadis, 2002; Ardaiz et al., 2003; Mikic-Rakic and Medvidovic, 2004), there is no explicit representation of the system-wide software architecture as it is partitioned amongst the K-Components in the system. The software architecture of a distributed K-Component application is *decentralised*. 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. External components update the local AMM by transferring their descriptions on connector binding (see figure 3.5).

To support self-adaptation, K-Components supports the specification of adaptation logic for monitoring and reconfiguring the AMM (see section 3.7). Reconfigurations are constrained to performing type-safe connector binding and component replacement operations. To support the communication of adaptation conditions between K-Components, an event-based architectural style allows the asynchronous communication of feedback events between connected K-Components. Two K-Components are connected when a component residing in one K-Component is bound to a component residing in the other K-Component. Feedback events enable the construction of coordination protocols between connected K-Components in decentralised environments.

In summary, the decentralised architectural style for K-Components partitions the system's software architecture across AMMs in the different K-Component runtimes, provides support for constrained self-adaptation through AMM monitoring and reconfiguration and supports the coordination of components using an event-based architectural style.

3.6.3 Definition of the Architecture Meta Model

The basic element of the AMM is a configuration graph of component interfaces, components and connectors. They are reified in the AMM as a typed, directed configuration graph, where component interfaces are the nodes, components are the type labels of nodes and connectors are edges.

Definition 3.4: the AMM configuration graph for the K-Component in runtime k is $M_k = \{c_{root_k}, L_k, W_k\}$, where

- M_k is the AMM in runtime k and $M_k \in \mathcal{M}$, the set of all AMMs in the system
- c_{root_k} is the root component associated with runtime k
- $L_k = \{l_1, .., l_M\}$ is the set of connectors in runtime k

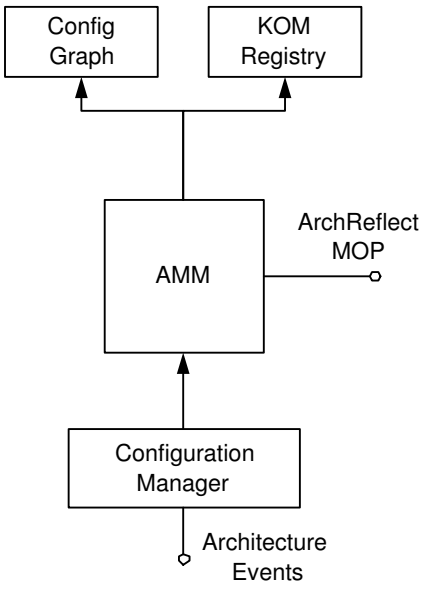


Figure 3.6: ArchReflect MOP uses the configuration graph and the KOM registry.

- $W_k = \{< i_1, c_1 >, ..., < i_N, c_N >\}$, where $i_{1..N} \in I_k \wedge (c_j = \emptyset \vee c_j \in \mathcal{C},\ j = 1..N) \wedge (\forall w_i \in N_k : w_k \in l_k \in L_k)$. W_k is the set of component interface/component pairs that represent nodes in the graph. Each component/interface pair is the target of some connector in the AMM, and c_j may or may not be resident in runtime k. Where the component entry in a node pair is empty, the connector to that node is not bound to a component.

3.6.4 ArchReflect MOP and ArchEvents

The *ArchReflect* MOP is a meta-level interface that allows the introspection and adaptation of the AMM and its components and connectors (see figure 3.6). ArchReflect uses the AMM's configuration graph and the KOM registry to call the reflection operations defined on components and connectors from sections 3.4.2 and 3.5.2, respectively. A set of architectural adaptation operations are defined in ArchReflect that allow the binding of connectors to components, the unbinding of connectors from components and the replacing of components.

ArchEvents is an interface that allows the updating of the set of components and connectors registered in the AMM (see figure 3.6). The ArchEvents interface is called by code in component and connector hooks defined at architectural reification points such as creation, deletion, binding and unbinding. The ArchEvents interface is a facade that delegates the registration and deregistration operations to the AMM's configuration graph and the KOM registry. ArchEvents represents the synchronous reflection interface to the AMM.

Internal AMM Operations

There are two internal introspection operations defined on the AMM to acquire references to components and connector objects using a logical identifier. They are used by implementations of the operations defined in ArchReflect and ArchEvents interfaces.

- $getComponent(id) \rightarrow c_i,\ c_i \in C_k$, returns a reference to c_i using id.

- $getConnector(id) \rightarrow \emptyset \vee l_i, where l_i \in L_k$, returns a reference to the connector object in runtime k.

ArchReflect MOP: Architectural Adaptation Operations

The ArchReflect MOP provides introspection and adaptation operations on the AMM. The reflective operations defined on components and connectors can be invoked using ArchReflect, but there are also a set of architectural adaptation actions, *bind/unbind/replace*, defined on ArchReflect that manipulate the AMM's configuration graph of components and connectors. The architectural adaptation actions meet the structural integrity and mutual consistency parts of the adaptation consistency requirements from section 2.3.3. Structural integrity constraints are maintained by the type-safe rebinding of connectors to components i.e., a component needs to provide the interface expected by a connector for a bind operation to succeed. To ensure mutual consistency during architectural adaptation, each connector implements a reconfiguration protocol that ensures connectors and components are passive before reconfiguration (see section 5.3). The maintenance of application state invariants, through automating component state transfer to a replacement component, is not supported by the model, but can be achieved through programmers providing a copy constructor that is used to instantiate the replacement component.

- $bind(l_i \times c_j) \rightarrow l_i, \ where \ \ getProvides(c_j) = getProvides(Target(l_i))$, binds a connector to a component and ensures structural consistency by only binding the connector to a type-compatible component.
- $unbind(l_i) \rightarrow l_i$, unbinds a connector from its target component.
- $replace(c_i \times c_j) \rightarrow c_j, where \ \ c_i \in C_k \wedge (getProvides(c_i) = getProvides(c_j) \wedge (Passive(l_i) \wedge Passive(c_i))$, replaces an old component c_i with a replacement component c_j and preserves structural consistency by ensuring the replacement component is type-compatible. It also meets the mutual consistency requirement by ensuring that the old component, its outgoing and its incoming connector are passive (Wermelinger, 2000) before reconfiguration starts. $Passive(l_i)$ implements reconfiguration protocol that ensures l_i, which is the incoming connector for c_i, meets the RPC-consistency requirements from section 3.5.1.

ArchEvents Interface: Registration/Deregistration Operations

Components and connectors are reified in the AMM by interceding in component and connector creation and synchronously registering component and connector instances in the AMM. The registration and deregistration operations that are defined on the ArchEvents interface are:

- $registerConnector(M_k \times l_i) \rightarrow M_k$, registers a connector with the AMM.
- $deregisterConnector(M_k \times l_i) \rightarrow M_k$, deregisters a connector from the AMM.
- $registerLocalComponent(M_k \times c_i) \rightarrow M_k$, registers a local component with the AMM.
- $deregisterLocalComponent(M_k \times c_i) \rightarrow M_k$, deregisters a local component from the AMM.

- $registerRemoteComponent(M_k \times i_i) \rightarrow M_k$, registers a remote component with the AMM using a its component descriptor.
- $deregisterRemoteComponent(M_k \times i_i) \rightarrow M_k$, deregisters a remote component from the AMM.

The feedback event registration and deregistration operations are also defined on ArchEvents, but the implementation of the operations is delegated to the AMM. Feedback events e_{local} and e_{remote} are defined in section 3.7.5.

- $registerLocalFeedbackEvent(M_k \times e_{local}) \rightarrow M_k$, registers a local feedback event object with the AMM.
- $deregisterLocalFeedbackEvent(M_k \times e_{local}) \rightarrow M_k$, deregisters a local feedback event object from the AMM.
- $registerRemoteFeedbackEvent(M_k \times e_{remote}) \rightarrow M_k$, registers a remote feedback event object with the AMM.
- $deregisterRemoteFeedbackEvent(M_k \times e_{remote}) \rightarrow M_k$, deregisters a remote feedback event object from the AMM.

3.7 Adaptation Contract Description Language

In K-Components the language used to specify a system's self-adaptive behaviour is separate from the programming language used for component programming. The Adaptation Contract Description Language (ACDL) is a declarative language for specifying adaptation logic for components and connectors in a runtime. This section is not intended to be a programming manual for the ACDL, but rather to describe its main concepts and discuss design decisions taken in developing the features of the language.

3.7.1 ACDL Overview

The ACDL provides programmers with a few high level abstractions with which they can specify a K-Component's self-adaptive behaviour. The abstractions are intended to allow programmers to declaratively specify self-adaptive behaviour as conditions on component or connector feedback state that trigger the execution of adaptation actions.

The main abstraction used to encapsulate adaptation logic in the ACDL is the adaptation contract. An adaptation contract is used to define a decision policy. A decision policy performs operations to monitor system state and contains conditions under which adaptation actions are performed. The ACDL provides component and connector feedback states, adaptation actions, feedback events and handlers as the abstractions a programmer can use to reason about adaptation conditions and perform adaptation actions on components and connectors.

Decision policies can be written in adaptation contracts as action policies or learning policies. An action policy can be described using both `if-condition-then-action` rules and the event-condition-action (ECA) paradigm. `If-then` rules define predicates on feedback states that when matched

cause the execution of adaptation actions. The ECA approach is based on declaring feedback events and handlers that get executed when the event is raised. Reinforcement learning and collaborative reinforcement learning policies can also be defined in the ACDL. These unsupervised learning policies require the definition of a reward model for adaptation actions in components, as well as the definition of a system's (or component's) self-adaptive behaviour as a Markov Decision Process.

3.7.2 Example Adaptation Contract in the ACDL

In this section an example of a self-adaptive file storage component is introduced to give an overview of the programming model for the ACDL, and some examples of its expressiveness. The component, `FileStorage`, provides an interface, `File`, and a component state `load` and is defined in table 3.1. An implemented and more complete version of this example is discussed in chapter 6.

```
// IDL
typedef sequence<octet> BinaryFile;
interface File {
   double submit(in string name, in BinaryFile contents);
   Binaryfile retrieve(in string name);
};
// K-IDL
component FileStorage {
  provides File;
  uses File n1;
  state load;
};
```

Table 3.1: Example FileStorage component in K-IDL.

```
handler repair_connector {
   component S2 = discover(FileStorage);
   connector c1(FileStorage::file::c_n1);
   rebind_connector(c1, S2);
   jitter(10000);
}
outgoing storageNotification (FileStorage::File::c_n1) {

   connector c1(FileStorage::File::c_n1);
   if (poll_state(c1,status)==CONNECTOR_BROKEN) {
       repair_connector();
   }
  state load(FileStorage);
  predicate  high_load(f:\\repository\predicates\repair_connector.xml);
   event adapt_high_load(load, high_load, Low, repair_connector);
}
```

Table 3.2: Example action policy in the ACDL.

```
<?xml version='1.0' encoding='utf-8' ?>
<dsg:cb-values xmlns:dsg='http://www.dsg.cs.tcd.ie'
                xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
                xsi:schemaLocation='http://www.dsg.cs.tcd.ie
            F:\\repository\\Schemas\\KEventDesc.xsd'>
        <dsg:unary-op dsg:op-name='12' dsg:value1='10'/>
</dsg:cb-values>
```

Table 3.3: Example predicate descriptor for a feedback state.

The self-adaptive behaviour for `FileStorage` is defined in table 3.2 using the ACDL. The table contains both a handler, containing a modular unit of reusable adaptation logic, and an adaptation contract that specifies both an `if-then` rule and defines an `event-condition-action`. The handler, `repair_connector`, rebinds a connector by firstly acquiring a reference to a component that implements the component interface `FileStorage` using a discovery service. Then it acquires a reference to a connector from the AMM using the connector identifier, `FileStorage::File::c_n1`. The connector is then rebound to the newly discovered component and the `jitter` operation ensures that no further reconfiguration operation is attempted for 10000 msec if the `rebind_connector` operation successfully completes.

The adaptation contract is then defined as a contract that will be associated with an outgoing connector, `outgoing`, and an identifier for the outgoing connector, `FileStorage::File::c_n1`, is specified as a parameter. In the contract, a reference to a connector object is then acquired from the AMM using the same declaration as in the handler code, and its status is monitored for a broken connection. If the connection is broken, the handler is invoked, attempting to rebind the connector to a new component. A feedback event is also defined as a predicate on the component state `load`, and invokes the handler if the predicate is matched. The predicate for event notification (see table 3.3) is matched when the value of the component state `load` increases above 10, as op-name '12' corresponds to the `greater-than` predicate.

This example program written in the ACDL shows how action policies encapsulate adaptation logic for a component and can be specified declaratively as rules and ECAs. The ACDL syntax is similar to C++ and it separates the specification of a component's self-adaptive behaviour from the component implementation code.

3.7.3 ACDL Features

This section introduces the different constructs and features in the ACDL that can be used by action policies.

ACDL Modules

The ACDL supports the declaration and definition of adaptation contracts and handlers as modular units of adaptation logic. An adaptation contract is associated with a single connector in the runtime. The syntax for declaring an adaptation contract is:

```
[outgoing | incoming]  <contract_name> (<connector_id>){
```

```
    [statement]*
};
```

A contract can be defined on either an incoming or outgoing connector to a component, and is defined as either `incoming` or `outgoing`.
Handlers are reusable functions that contain adaptation logic and can be executed in response to feedback events being raised or called from an adaptation contract.

```
handler <handler_name>() {
  [statement]*
}
```

ACDL Variables

The ACDL supports the declaration of variables that represent connectors, components, and component states in both adaptation contracts and handlers:

```
connector <connector_name>(<connector_id>);
component <component_name>(<component_id>);
state <state_name>(<component_name>);
```

A component identifier, `<component_id>`, is used to identify the component in the runtime, where `<component_id>` represents the component's class name or the component's unique identifier generated by the runtime. A connector identifier, `<connector_id>`, is of the form

```
<component_id>::<uses_interface>::<connector_name>
```

and `<state_name>` refers to a state declared on a component interface.
The ACDL also supports the definition of feedback events in adaptation contracts. Notification conditions for feedback events are defined in a predicates descriptor file, defined in XML:

```
predicate <pred_comp_state_xml>(XML_File);
event <event_name>(<state_name>,<pred_comp_state_xml>,Priority,<handler_name>);
```

Standard ACDL Functions

The ACDL supports the monitoring of component feedback states in both handlers and adaptation contracts. Feedback state queries can be executed using:

```
double poll_state(<state_name>);
```

The ACDL supports the execution of adaptation actions in both handlers and adaptation contracts. Component adaptation actions are executed using:

```
double action(<action_id> [, strategy]);
void resolve_invoke(<action_id> [, Priority [, strategy]);
```

where <action_id> is of the form <component_id>::<action_name> and double is a scalar reward supplied by either the AMM or a component indicating the success of the action. A strategy is a scalar that can also be passed as a parameter to an adaptation action. It is used by component adaptation actions to supply information about how the component adaptation action should be executed, e.g., which adaptation strategy to perform. There are two different ways of invoking component defined actions, via the AMM using action and via a conflict resolver component using resolve_invoke. Component defined actions invoked using action return a double (a scalar) as a result. This value is used by learning policies as a model for a reward. The resolve_invoke method does not return a scalar, as it executes the adaptation action asynchronously by placing an invocation request in the queue of a conflict resolver component (see section 5.7.5). The conflict resolver is an autonomous component that stores a queue of adaptation actions, traverses the queue looking for conflicting actions and then resolves any conflicts and executes the actions. It uses the Priority parameter to resolve conflicts between adaptation contracts stored in its queue, but not yet invoked. Other architectural reflection operations are defined in the ArchReflect MOP and have the following signature:

```
double ArchReflect::<operation_name>([param[, param]*]);
```

A jitter declaration is also provided in the ACDL that prevents adaptation actions from being executed for a programmer specified period of time. This can be used to help dampen overly frequent adaptations in "gray-zones", the borderline areas where a system switches from one configuration to another. In setting the jitter time, there is a trade-off between reducing the potential for system hysteresis and the lack of certainty that the system is in the optimal configuration. A jitter operation can also be encapsulated inside the a catch section of a try-catch statement, and if an exception is thrown the user-supplied function object, <fn_object>, is executed. In this case, the jitter statement can be used to specify the conditions under which reconfiguration operations that fail are retried, e.g., retry the reconfiguration operation a fixed number of times or after a specified timeout.

```
jitter(<millisecs>[, <fn_object>]);
```

Standard ACDL Features

The ACDL also supports standard language features such as expressions, primitive types, logical operators, conditionals. Exception handling is also required as many errors cannot be checked at compile time and require handling at runtime.

```
try { [statement]* } catch { [statement]*}
double | string | bool
&& | || | ! | > | >= | < | <= | between
if <condition> {  [statement]* } [else { [statement]* } ]
```

There is no support for while loops in the ACDL, as adaptation contracts should complete execution in bounded time to avoid locking shared resources indefinitely.

3.7.4 Adaptation Contracts

An adaptation contract is a set of declarations, expressions, monitoring operations and adaptation actions defined in the ACDL. Adaptation contracts provide a 1st party decision policy for components in the runtime and are compiled into autonomous, reflective programs concerned with monitoring, adapting and coordinating components. Adaptation contracts are examples of software agents (Wooldridge and Jennings, 1995) and have similar goals to autonomic management components found in other systems, such as Kephart's autonomic manager (Kephart and Chess, 2003). Adaptation contracts aim to reduce the amount of component management actions that must be performed by administrators, thus meeting one of the goals of autonomic computing.

Adaptation contracts are associated with connectors in the runtime, as connectors provide intercession points at which they can be initialised, started and stopped. An adaptation contract is associated with either an incoming or an outgoing connector by declaring it using the keyword `incoming` or `outgoing` and passing a connector identifier as a parameter to its constructor. If an adaptation contract is associated with an incoming connector, it is effectively associated with a single component in the runtime, as there is only a single incoming connector per component. The adaptation contract is initialised and started when the incoming connector is created. Components, however, may have many outgoing connectors and, therefore, many outgoing contracts. An outgoing adaptation contract is associated with an outgoing connector, again, by passing a connector identifier as a parameter to its constructor. Outgoing connectors, however, may load and unload different adaptation contract objects at runtime as adaptation contract objects are created and destroyed on connector binding and unbinding operations. When an outgoing connector binds to a remote component, it dynamically loads a proxy to the remote object as a shared library. Adaptation contract objects are typically deployed in these proxies and are loaded and started when the outgoing connector binds to the remote component. When the connector unbinds from the remote component, the adaptation contract object is shutdown and destroyed. This way, adaptation contract objects can be replugged to update the self-adaptive behaviour of a K-Component at runtime. An adaptation contract is defined as either of the following:

```
incoming <contract_name> (<connector_id>) { ....}
outgoing <contract_name> (<connector_id>) { ....}
```

In an adaptation contract, a programmer can declare variables that refer to components, connectors and component feedback states in the AMM, as well declare variables using the primitive types supported. The references to the component and connector instances are untyped. They refer to a generic component and connector types defined in the K-Component framework. This prevents ACDL programmers from invoking methods defined on the public interface of components or connectors, which would require component programmers to handle concurrent access to components by adaptation contracts, breaking the separation of concerns. Programmers can, however, adapt a K-Component by executing adaptation actions defined on the ArchReflects MOP.

Definition 3.5: an adaptation contract for component c_j in runtime k is $n_{ij} = \{l_i, M_k, Policy_i\}$, where

- n_{ij} is the adaptation contract i associated with connector $l_i$from the set of adaptation contracts, N_k, in runtime k.
- l_i is either an incoming or outgoing connector for component c_j , i.e., $getTarget(l_i) \rightarrow c_j \vee getSource(l_i) \rightarrow c_j$
- M_k is the AMM in runtime k and $M_k \in \mathcal{M}$.
- $Policy_i$ is the set of decision policies defined in n_{ij} i.e., all the adaptation logic defined in the contract.

Adaptation contracts are registered using the configuration interface (see figure 3.1) with the *adaptation contract manager*, an autonomous singleton that is responsible for managing the lifecycle of adaptation contracts:

- $registerAdaptationContract(N_k \times n_{ij}) \rightarrow N_k$, registers an adaptation contract with the adaptation contract manager.
- $unregisterAdaptationContract(N_k \times n_{ij}) \rightarrow N_k$, registers an adaptation contract with the adaptation contract manager.

3.7.5 Feedback Events

Feedback events are used to specify an ECA policy in the ACDL. A feedback event is defined as a predicate on the value of a component feedback state and an associated handler. Feedback events are raised when their predicate is matched, and this triggers the execution of a handler that contains adaptation logic. A feedback event definition consists of a component feedback state, a predicate defined on the feedback state, a priority and a handler. The ACDL supports the definition of feedback events and the declaration of predicates in adaptation contracts, using the following syntax:

```
predicate <pred_comp_state>(<XML_file>);
event <event_name>(<state_name>,<pred_comp_state>,Priority,
                   <handler_name>[, <connector_id>]);
```

A predicate contains the conditions under which a feedback event is raised. They are defined externally to the ACDL in a *XML predicate descriptor* that contains a set of predicates on the value of the component feedback state. The list of predicates on component feedback states that are supported is defined in table 3.4. Predicates can be defined on the current value of a feedback state, as well as changes in the value of a feedback state, e.g., value-changed and rate-of-change. Two of the predicates provide custom support for the specification of a CRL policy in the ACDL. These are the specification of a rate of decay for cached component feedback states values and an advertisement predicate that periodically notifies registered clients of the current value of a feedback state.

Feedback event communication is managed by a *feedback event manager*, a component deployed in every K-Component runtime. Feedback event managers act as both producers and consumers of feedback events and are connected using an asynchronous event style (see section 3.5.1). Feedback events that refer to remote components are known as *remote feedback events*, while feedback events

Predicate	Type	Parameters
LESS-THAN	Unary	value
GREATER-THAN	Unary	value
EQUALS	Unary	value
NOT-EQUALS	Unary	value
IS-TRUE	Unary	boolean
DECAY	Binary	(td,scaling_factor)
VALUE-CHANGED	Binary	(boolean,old_value)
INSIDE-RANGE	Binary	(value,value)
OUTSIDE-RANGE	Binary	(value,value)
RATE-OF-CHANGE	Binary	(rate,old_value)
ADVERTISEMENT	Binary	(period,timeout)

Table 3.4: Predicates on component feedback states.

that refer to local components are known as *local feedback events*. The following definitions of local and remote feedback events are provided, as they are required for the definition of a CRL policy in 4.2.8:

Definition 3.6: A remote feedback event registration object is
$e_{remote_k} = \{f_{ij}, p_k, FM_k, l_{c_i}\},\ e_{remote_k} \in E_{remote_k}$ where

- E_{remote_k} is the set of remote feedback events defined in runtime k.
- f_{ij} is a component feedback state f_i defined on a remote component c_j that is defined in M_k, $c_j \notin C_k \wedge c_j \in W_k \in M_k$.
- $p_k \in P_k$ is a predicate on component feedback state f_{ij}.
- FM_k is a reference to the local feedback manager used for event notification.
- l_{c_i} is the outgoing connector deployed in $c_i \in C_k$ that is used to register the event in the remote feedback event manager containing the component c_j.

Definition 3.7: A local feedback event registration object is
$e_{local_k} = \{f_{ij}, p_k, priority, handler_{ij}\},\ e_{local_k} \in E_{local_k}$ where

- E_{local_k} is the set of feedback events defined in runtime k.
- f_{ij} is a component feedback state i defined on a local component, $c_j \in C_k$.
- $p_k \in P_k$ is a predicate on component feedback state f_{c_j}.
- $priority \in \{low, normal, high, system\}$.
- $handler_{ij}$ is a handler defined in adaptation contract n_{ij} in runtime k.

Definition 3.8: A remote feedback notification object sent by FM_k is
$e_{notify_k} = \{f_{ij}, FM_o, v\},\ e_{notify_k} \in E_{notify_k}$ where

- E_{notify_k} is the set of feedback notification objects that can be sent by feedback manager, FM_k, in runtime k.
- FM_o is the remote feedback manager to which the feedback notification object, e_{notify_k}, is sent when the event is raised.

- f_{ij} is the component feedback state on $c_j \in C_k$ to which the feedback notification object refers.
- $v \in \mathbb{Z}$, the value of the component feedback state, f_{ij}.

3.7.6 Reinforcement Learning Policy

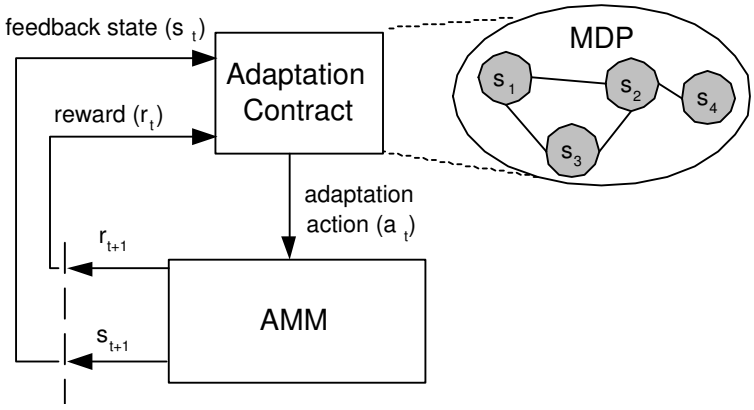


Figure 3.7: Reinforcement learning policy in K-Components.

Reinforcement learning (RL), introduced in section 2.4.3, is an unsupervised learning technique that can be used to enable an adaptation contract to learn its self-adaptive behaviour at runtime. There is declarative support in the ACDL for specifying a RL policy in an adaptation contract.
In the model for the RL policy, the adaptation contract represents the RL agent that attempts to optimise its interaction with its environment (see figure 3.7). The environment of the agent is modelled as a Markov Decision Process (MDP) that consists of states actions, and provides a reward model. In K-Components the environment of an adaptation contract consists of the AMM and its constituent components and connectors. States, actions and rewards in a RL model for an adaptation contract are defined in terms of components and connectors. The goal of the RL policy is to learn an optimal decision policy for an adaptation contract. RL policies are useful for components in a decentralised system, as they allow components to cope with uncertainty in their environment by enabling components to learn a decision policy for how to adapt and optimise their behaviour in their environment.

MDP in the ACDL

In order to define a RL policy in the ACDL, the RL problem has to be defined as a MDP. A MDP is defined as a set of states, actions, a state transition model and a reward model. There is a natural mapping between MDP states and the component feedback states in the K-Component model. Component feedback states can be used to define MDP states that define the dynamic behaviour of the component, but designers should ensure that component feedback states used as MDP states satisfy the Markov Property.

Component feedback states are represented as scalars, but MDP states are represented as booleans in a system, with only one of the MDP states in the system evaluating to true at any given time. A component programmer that wishes to model behaviour of a component as a MDP using component

feedback states can represent the scalar as a boolean, i.e., as a 1 or 0, but has to ensure that the constraint of only a single active feedback state at any given time is met. This constraint cannot be enforced at compile-time in the ACDL.

It may also be the case that a programmer wishes to define the dynamic behaviour of a component in terms of the value of a single component feedback state, e.g., partition the values of the feedback state into regions, each of which represents a MDP state. As a result of this potential need, the ACDL allows a programmer to define a MDP state as a variable in an adaptation contract. A MDP state can then be defined as a predicate on the feedback state of a component or a connector using the format:

```
rl_state <rl_state_name>(<state_name> [,predicate]);
```

The set of states in the MDP consists of predicates on the feedback state(s) of the adaptation contract's associated component. Predicates can be defined using the predicate's descriptor, from section 3.7.5, or left empty. An empty predicate indicates that the component feedback state will be represented as a boolean, whereby the predicate is matched if the value of the feedback state is 1, otherwise it is not matched.

Definition 3.9: A MDP state is defined as $s_i = \{f_{ij}, p\}$ where

- $s_i \in S_i$ is a MDP state that represents a predicate on the value of a component feedback state f_{ij}. The state should satisfy the Markov Property.
- S_i is the set of all states defined in the MDP, for which only a single state can be active at any instant in time.
- p is a predicate on component feedback state f_{ij}. Only one predicate from the set of predicates, P_i, defined on the component feedback states in the MDP should match at any given time.

Actions are also a required part of the MDP. In the ACDL, the set of adaptations actions defined on the adaptation contract's associated component as well as the architectural adaptation actions can be included as actions in the MDP. A state transition model is also required for the MDP. This can be provided as a state transition distribution function or not provided, and learnt by a model-free RL algorithm.

Reward Model

A reward model is required by a RL policy. In the ACDL, the reward model is used as a metric to quantify the success of adaptation actions. Rewards (or reinforcements) are typically represented as scalars and are supplied by the environment of an adaptation contract after the execution adaptation action:

```
double reward = ArchReflect::action(<actionID>);
```

The reward model for a K-Component, R_k, consists of the set expected rewards for executing all adaptation actions $\mathcal{A}$ in all system states $\mathcal{S}$.
A reward for executing an adaptation action has to be calculated using some information about the dynamic behaviour of the system. A reward model for a component adaptation action has to be

specified by the component programmer. Typically rewards are generated from component feedback state values that describe the dynamical state of the component. For example, consider a component that has a feedback state that describes the operating state of the component as optimal, suboptimal or failed and an adaptation action to modify the behaviour of the component. The adaptation action may be invoked by an adaptation contract and in order to calculate the reward for its execution, it can firstly observe the feedback state of the component (as optimal, suboptimal or failed), secondly execute the adaptation action code, and then finally observe the feedback state of the component again. Any change in the operating state of the component from before the execution of the adaptation action code to after the execution of the adaptation action code can be used to calculate a reward. The reward is calculated as a scalar and is passed as a return value to the adaptation contract that invoked the adaptation action. Components can also support a *null adaptation action* that returns a reward in the case where no adaptation action is taken. Null adaptation actions are necessary for learning policies that repeatedly execute adaptation actions, since the optimal policy is often not to adapt a component.

Definition of a MDP

There is no support for defining a MDP in the ACDL. However, MDPs are defined as part of a RL policy and a definition is included here as a prerequisite for the definition of a RL policy.

Definition 3.10: A MDP is $mdp_i = \{n_{ij}, S_i, A_i, R_k, T_i\}$ where

- mdp_i is a MDP in adaptation contract n_{ij} in runtime k.
- $S_i = \{s_1, ..., s_N\}$, where S_i is the set of MDP states in mdp_i.
- A state s_i is $s_i = \{f_{ij}, p_j\}$, where $p_j \in P_i$ is a predicate on a component feedback state f_{ij}. A state is active if the predicate for its component feedback state is matched.
- A_i is the set of available adaptation actions in adaptation contract n_{ij}. This includes the set of component adaptation actions defined on c_i and the set of architectural adaptation actions defined on M_k.
- R_k is the reward model in runtime k is a function $R_k : S \times A \rightarrow \mathbb{R}$. $R(s, a)$ is the expected instantaneous reinforcement from action a in state s.
- T_i is the state transition distribution function, $S_i \times A_i \rightarrow \Pi(S_i)$, defined on mdp_i. $\Pi(S_i)$ is the set of probability distributions over the set S_i of component feedback states defined on c_i.

The actions that are available in the different MDP states can be limited by defining a constraint:

- $AvailableActions : s_i \rightarrow A(s_i), \quad A(s_i) \subseteq A_i, s \in S_i$

Such a function is useful to enable component providers determine the current state when building a reward model, $R_i : S_i \times A_i \rightarrow \mathbb{R}$, for a component adaptation action a. Given a state s, calculated using a component feedback state defined on c_i, the set of adaptation actions available in s can be constrained to the set of component adaptation actions, A_i, defined on c_i. This enables the provider

of the component to calculate the value $R(s, a)$ in adaptation action a, since it can observe the current active MDP state s by examining the value of the component feedback states, F_i.

Definition of a RL Policy

A RL policy is defined in an adaptation contract using the following format:

```
ListStates <list_states> = {<rl_state_name1>,..,<rl_state_nameN>};
ListActions <list_actions> == {<actionID1>,...,<actionIDN>};
rl_policy <agent_name>(<list_states>,<list_actions>,RLAlgorithm);
start_mdp(<agent_name>, <num_trials>=infinite);
```

A RL policy is defined as a MDP and a RL algorithm. There is no support for constraining the set of actions allowed in particular states or the explicit definition of a state transition model. A state transition probability matrix is provided by default and must be updated by the learning algorithm. A RL policy can be executed by a call to `start_mdp` in an adaptation contract. The behaviour of an adaptation contract executing an RL policy is as follows: the adaptation contract observes its set of states for the current MDP state and then calculates the adaptation action to execute using its policy. After executing the adaptation action, the system makes a state transition and the adaptation contract receives a reward, which is then used by a RL algorithm to optimise its policy. The adaptation contract again observes its current state and follows its policy thereafter until the MDP reaches a terminal state or the specified number of trials has been exceeded. An example of a learning policy in an adaptation contract is provided in chapter 6.

Definition 3.11: A reinforcement learning policy is $RL_i = \{mdp_i, alg_{RL}, maxTrials\}$ where

- $RL_i \in Policy_i$ is the reinforcement learning policy defined in adaptation contract n_{ij}.
- alg_{RL} is the reinforcement learning algorithm.
- $maxTrials$ is the maximum number of times actions should be attempted until the RL policy terminates.

3.8 Summary

This chapter has presented the K-Component model at an abstract level. The K-Component model provides a structured model for building self-adaptive software that operates in a decentralised environment. The model uses architectural reflection to reify the structure of a K-Component as an architecture meta-model. Information concerning adaptation conditions in components and connectors is available in the architecture meta-model in the form of component and connector feedback states. Reflective programs, called adaptation contracts, can be written that are concerned with monitoring feedback states, identifying adaptation conditions and adapting the components and connectors at runtime. An asynchronous model of reflection is used to execute adaptation contracts asynchronously to system operation. This enables adaptation contracts to continuously and proactively reason about

and adapt system operation. This is particularly useful in the case where adaptation conditions arise in sources external to the system.

A declarative programming language, the ACDL, is used to aid programmers in the specification of the decision policy for an adaptation contract. The decision policy can be specified as an action policy or a learning policy. Action policies can be declared using rule-based or event-condition-action approaches. However, when the space of states and actions is too large for programmers to handle or where programmers cannot know in advance the likely affects of adaptation actions, a RL policy can be defined to help adaptation contracts learn their self-adaptive behaviour.

The autonomic properties that can be supported by applications built using the K-Component model are constrained by both the set of feedback states and feedback events that can be monitored and the range of adaptation actions that can be performed on components and connectors. The ACDL supports two different types of adaptation actions, architectural adaptation actions that reconfigure the architecture meta-model and component adaptation actions defined on component interfaces. Architectural adaptation actions are useful for reconfiguring connections to faulty or poorly performing components. Component adaptation actions can be used to adapt the behaviour of a component, e.g., to optimise component performance by changing the implementation strategy of an internal algorithm in the component. The availability of a reward model for component adaptation actions and architectural adaptations is used as the basis for evaluating adaptation actions and learning a decision policy.

While K-Components can be used to build self-adaptive components, the construction of autonomic distributed systems using self-adaptive components requires that those components coordinate their self-adaptive behaviour to meet system-wide goals. The next chapter addresses the problem of coordinating K-Components in order to establish and maintain system-wide properties in decentralised systems.

Chapter 4

Collaborative Reinforcement Learning

"The irreversibility [of time] is the mechanism that brings order out of chaos."

Ilya Prigogine, Order Out of Chaos (1984)

This chapter describes one of the main contributions of this thesis, the collaborative reinforcement learning (CRL) algorithm. CRL addresses the requirement of a distributed autonomic system for a coordination model that can establish and maintain system properties in decentralised environments. CRL is an extension to RL for decentralised multi-agent systems and provides a decentralised coordination model similar to those found in swarm intelligence algorithms (Kennedy and Eberhart, 2001), where agents collectively learn from the successes of their neighbours. In CRL, individual agents maintain a local, partial model of the system that includes information about their neighbouring agents. The system properties of the algorithm are a product of the collective behaviour of agents that coordinate using converged local models.

This chapter represents a change in style from the previous chapter. The terminology used to describe CRL algorithm is consistent with the terminology from the RL literature (Sutton and Barto, 1998; Kaelbling et al., 1996; Littman and Boyan, 1993; Sutton, 1988; Barto and Mahadevan, 2003). In particular, the term agent is used to describe autonomous decision making entities in a system that executes and learn a 1^{st} party decision policy.

4.1 Decentralised Coordination and Autonomic Computing

Carriero and Gelernter provide a definition for coordination in (Gelernter and Carriero, 1992):

"Coordination is the process of building programs by gluing together active pieces"

where active pieces can be processes, autonomous objects, agents or applications. Coordination is the logic that binds independent activities together into a collective activity. Coordination models have been developed to describe the "glue" that connects computational activities. They can be based on centralised or decentralised coordination models. A system built using a centralised coordination model is a multi-agent system where the behaviour of its agents is controlled either by an active manager component or by a predetermined design or plan followed by the agents in the system

(Goldin and Keil, 2007). A system built using a decentralised coordination model is a self-organising multi-agent system (Goldin and Keil, 2007), whose system-wide structure or behaviour is established and maintained solely by the interactions of its agents that execute using only a partial view of the system.

Coordination models are necessary for the construction of autonomic distributed systems as they organise the autonomic and self-adaptive behaviour of individual autonomous components towards system goals. A lack of coordination among autonomous components in a distributed system can lead to interference between the different components' adaptive or autonomic behaviour, conflicts over shared resources, suboptimal system performance and hysteresis effects (Efstratiou et al., 2002b). For example, a distributed system that is composed of autonomic components, where components optimise their behaviour towards component goals is not necessarily optimised at the system-level, as there is the possibility that conflicting greedy decisions taken by components may result in sub-optimal resource utilisation or performance at the system-level. In the problem of decentralised resource allocation in autonomic systems, Boutilier motivates a decentralised approach to coordination in autonomic computing by stating that (Boutilier et al., 2003)

> "the reasoning required to support optimal resource allocation is necessarily distributed, thus requiring some form of cooperative negotiation among the computing elements that have conflicting needs for critical resources."

In order to optimally adapt a system to a changing environment the components must respond to changes in a coordinated manner, but in decentralised environments, the coordination mechanism cannot be based on centralised or consensus-based techniques for the reasons outlined in section 2.1.

Increasingly, researchers are investigating decentralised coordination approaches to establish and maintain system properties (Dorigo and Caro, 1999; Andrzejak et al., 2003; Ardaiz et al., 2003; De Wolf and Holvoet, 2003; Khare, 2003; Montresor et al., 2003; Dowling et al., 2004; Boutilier et al., 2003). Decentralised control is based on defining local coordination or control models for components that have only partial views of the system, support only localised interaction and have no global knowledge.

Components typically store locally a partial, estimated model of the system and interaction protocols defined between neighbouring components enable them to collectively improve the accuracy of their local, estimated models (Khare and Taylor, 2004; Curran and Dowling, 2004). This can, under certain conditions, result in convergence between the estimated models on a common view of the system or environment (Jelasity et al., 2003). Components that have converged models can coordinate their behaviour using their local models to perform collective adaptive behaviour that can establish and maintain system-wide properties. These system properties emerge from the local interaction between neighbouring components and with no explicit representation of system properties on the level of the individual component (Dorigo and Caro, 1999; Dowling et al., 2004; Andrzejak et al., 2003; Ardaiz et al., 2003).

Decentralised coordination techniques have been developed that are based on cooperation (Boutilier et al., 2003; Khare and Taylor, 2004) and competition (Panagiotis et al., July-August 2002) between components. Both approaches are typically evaluated by how they optimise some system property, such as an autonomic property of the system.

Some problems associated with decentralised models include the uncertain outcome of control actions on components, as their effect may not be observable until some unknowable time in the future. Also, optimal decentralised control is known to be computationally intractable (De Wolf and Holvoet, 2003), although systems can be developed where system properties are near-optimal (Littman and Boyan, 1993; Caro and Dorigo, 1998; Jelasity et al., 2003; Curran and Dowling, 2004), which is often adequate enough for certain classes of system. Finally, the design of decentralised coordination models is difficult, as there is no existing methodology to help translate a set of top-down requirements for system properties to a specification of the local coordination or control algorithms. As a result, experimentation plays a crucial role in verifying the establishment and maintenance of system properties (Dowling et al., 2005).

4.2 Collaborative Reinforcement Learning

CRL is an algorithm that can be used to build decentralised coordination models. CRL can be used to build decentralised systems with autonomic properties, such as the ability to adapt and optimise system behaviour to a changing environment. CRL models the desired system behaviour as a set of system optimisation problems that are solved by decentralised agents that learn how to interact with their environment using reinforcement learning (RL) (Sutton and Barto, 1998; Kaelbling et al., 1996) and coordinate their behaviour using positive and negative feedback. CRL agents are autonomous programs that store a view of their local environment as a cache of recent observations of their neighbours and their states. The set of agents in a CRL system is dynamic, with agents leaving and joining the system at runtime. Every agent has a dynamic set of neighbours, defined by the set of other agents found within its locality, where the definition of a locality is system-specific, but is often determined by some physical system limitation, e.g., the number of directly connected peers in a P2P system or the communication range in wireless networks. In K-Components, CRL agents are modelled as adaptation contracts (see section 4.2.8).

CRL extends RL with feedback models for its local view of the environment, including a negative feedback model that decays an agent's local cache and a collaborative feedback model that allows agents to exchange the values of their states. In a system of homogeneous RL agents, collaborative feedback enables agents to share state information that can increase convergence between local models. Since action selection by RL agents is based on their local system models, i.e., the agent's policy, collaborative feedback can, in certain circumstances, increase the probability of a group of agents taking the same or related actions. The collaborative feedback process can produce positive feedback in action selection probability over a group of CRL 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 collective behaviour in a group of agents (Camazine et al., 2003; Bonabeau et al., 1999). 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.

A typical evaluation criterion for system optimisation techniques is the amount of time required until the distributed policies converge to produce collective behaviour that meets the system optimisation

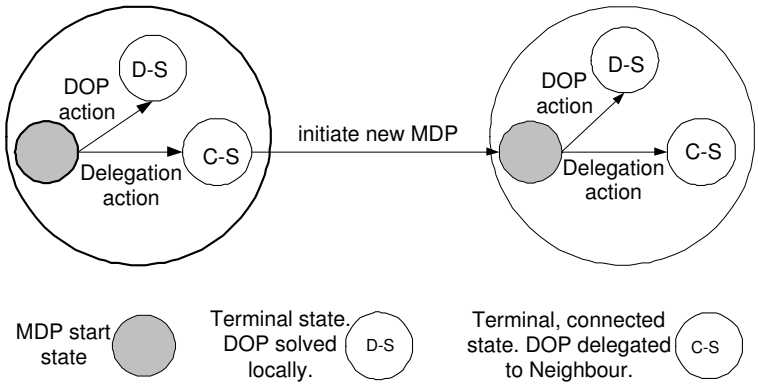


Figure 4.1: DOP and delegation actions in MDPs.

goals (Crites and Barto, 1998). However, in open, decentralised systems, the system's environment is dynamic and another evaluation criterion is the *collective adaptation agility* of agents (Li, 2000). Since optimal system behaviour may change with a changing environment, the collective adaptation agility of agents describes the ability of a group of agents to collectively adapt their behaviour to changes in their environment in order to continue to meet the system optimisation criteria.

4.2.1 Coordinating the Solution to Discrete Optimisation Problems

In CRL, system optimisation problems are decomposed into a set of discrete optimisation problems (DOPs) (Dorigo and Caro, 1999) that are solved by collaborating RL agents. There are many autonomic properties in distributed systems that can be naturally discretized into DOPs that can be distributed amongst agents in a distributed system, such as resource allocation over a group of servers (Ardaiz et al., 2003; Nowicki et al., 2004) and locating replicated resources in a service-oriented network (Andrzejak et al., 2003).

In a multi-agent system, the solution to each DOP is initiated at some starting agent in the network and terminated at some (potentially remote) agent in the network. Agents can either attempt to solve each DOP locally or find a neighbour that can solve the DOP. Each agent uses its own policy to decide probabilistically on which action to take to attempt to solve a DOP. Local decisions produce near-optimal system behaviour, although due to network dynamism and the lack of global knowledge, optimal system behaviour cannot be guaranteed. In CRL there are 3 types of action defined that can be executed by a CRL agent to solve a DOP (see figure 4.1). These are:

1. *DOP actions*, $\mathcal{A}_{p_i}$, try to solve the DOP locally at the agent
2. *delegation actions*, $\mathcal{A}_{d_i}$, are coordination actions that delegate the solution of the DOP to a neighbouring agent
3. *discovery actions* are coordination actions that an agent can execute in any state to attempt to find new neighbours.

An agent is more likely to delegate a DOP to a neighbour when it either cannot solve the problem locally or when the *estimated cost* of solving it locally is higher than the estimated cost of a neighbour solving it. In CRL, a group of agents coordinate their solution to a set of DOPs by taking independent

decisions about when to try to discover new neighbours and when to delegate the solution to a DOP to a neighbour, with the goal of optimising the solution to the set of DOPs at the system level.

4.2.2 Connected States for Delegating DOPs

In heterogeneous distributed systems, agents typically possess different capabilities for solving a given DOP. In the K-Component model, these capabilities are expressed via contractually specified interfaces. However, in RL the only abstractions available to an agent are states, actions and rewards. To model the differing capabilities of agents, CRL allows newly discovered agents to negotiate the establishment of *connected states* with their neighbours. For example, agents may exchange device capability information and use this information to determine whether or not they are able to establish a connected state. Connected states represent the contractual agreement between neighbouring agents to support the delegation of DOPs from one to the other.

Connected states map an *internal state* on one agent to an *external state* on at least one neighbouring agent. An internal state on one agent can be connected to external states on many different neighbouring agents. An agent's set of connected neighbours represents its *partial-view* of the system.

In CRL, for every neighbour, n_j, with whom agent n_i shares a connected state, there exists a delegation action $a_j \in \mathcal{A}_{d_i}$ that represents an attempt by n_i to delegate the solution to a DOP to n_j (see figure 4.1). If the delegation action is successful, n_i makes a state transition to its connected state s, terminating the MDP at n_i, and n_j initiates a new MDP to handle the DOP. For the case where an internal state on an agent maps to more than one external state, a delegation action may initiate a new MDP at more than one agent. Apart from an agent's capabilities, run-time factors, such as the agent's available resources and the quality of its network connections, also affect the ability of agents to solve a given DOP. These capabilities can be modelled in the agent's reward model.

4.2.3 Distributed Model-Based Reinforcement Learning

State transitions in CRL may be to a local state on the current agent or to a remote state on a neighbouring agent, via a connected state (see figure 4.2). In distributed systems, when estimating the cost of the state transition to a remote state on a neighbouring agent we also have to take into consideration the network *connection cost* to the neighbouring agent. For this reason, we use a *distributed model-based reinforcement learning algorithm* that includes both the estimated optimal value function for the next state at agent n_j, $V_j(s')$, and the connection cost, $D_i(s'|s,a) \in \mathbb{R}$ where $a \in \mathcal{A}_{d_i}$, to the next state when computing the estimated optimal state-action policy as $Q_i(s,a)$ at agent n_i (see equation 4.5).

In the distributed model-based RL algorithm, the reward model consists of two parts. Firstly, a *MDP termination cost*, $R(s,a) \in \mathbb{R}$, provides agents with evaluative feedback on either the performance of a local solution to the DOP or the performance of a neighbour solving the delegated DOP. Secondly, a connection cost model, $D_i(s'|s,a)$, provides the estimated network cost of the attempted delegation of the DOP from a local agent to a neighbouring agent. The connection cost for a transition to a state on a neighbouring agent should reflect the underlying network cost of delegating the DOP, while the connection cost for a transition to a local state after a delegation action should reflect

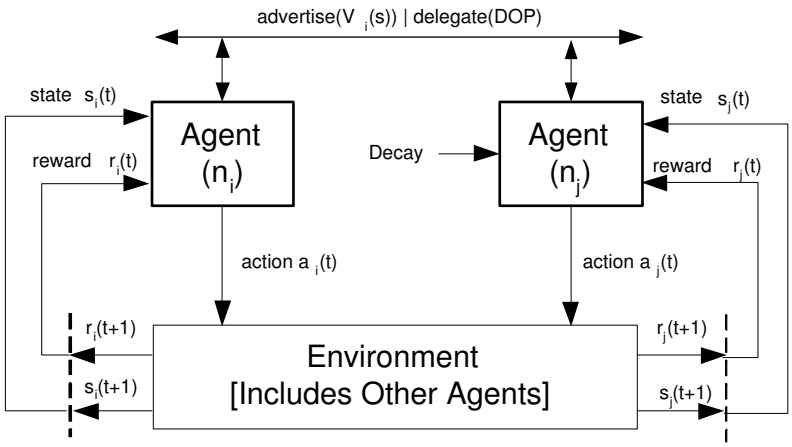


Figure 4.2: Operation of CRL.
In CRL, an agent, n_i, takes an action at time t, $a_i(t)$, and makes in a transition to state $s_i(t+1)$, that produces a reward, $r_i(t+1)$. The updated V values are advertised to neighbours, and the agent delegates a DOP to a neighbouring agent. Cached V values are decayed over time.

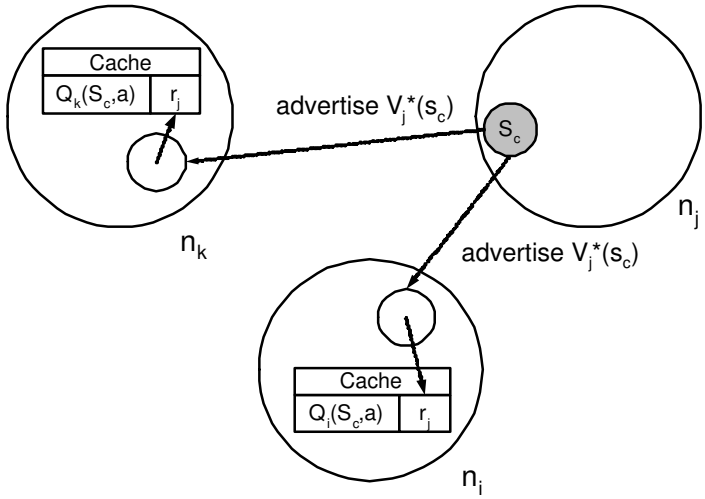


Figure 4.3: Advertisement of agent n_i's connected state value to the caches in agents n_j and n_k.

the cost of the failed delegation of the DOP. The connection cost model requires that the environment supplies agents with information about the cost of distributed connections as rewards. Ideally a middleware will provide support for monitoring connection quality, such as in (Atighetchi, 2003).

4.2.4 Local System Model and Advertisement

In CRL, each agent maintains a local model of its partial view of the system in a cache, $Cache_i$, that stores V-values for connected external states on neighbours. The cache consists of a table of Q-values, for all delegation actions, a_d, at connected state s in agent n_i, and the last observed $V_j(s)$ for the causally-connected state s at agent n_j, i.e., the cache stores the estimated cost of a neighbour n_j solving the DOP. A $Cache_i$ entry is a pair $(Q_i(s,a_j), r_j)$, where r_j is the cached $V_j(s)$ value.

Observations for $V_j(s)$ can come in two forms. Firstly, an agent can execute a delegation action, a_d, at connected state s in agent n_i and receive $V_j(s)$ as a synchronous reply from agent n_j. Alternatively, and more commonly, agent n_i is informed of changes to $V_j(s)$ at agent n_j using *advertisement* (see figure 4.3). When the agent n_i receives a $V_j(s)$ advertisement from neighbouring agent n_j for a

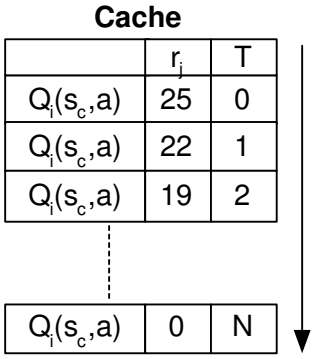


Figure 4.4: Decay of cached $Q_i(s_c, a)$ entries over time

connected state s, it updates r_j in $(Q_i(s, a_j), r_j)$. Examples of implementation strategies for V-value advertisement in distributed systems include periodic notification/broadcast/multicast, conditional notification/broadcast/multicast, and piggybacking advertisement in request/acknowledgement packets.

4.2.5 Decay of the Local System Model

Similar to RL, CRL models are based on MDP learning methods that require complete observability (Kaelbling et al., 1996), however at any given agent in a decentralised system the set of system-wide states are only partially observable. To overcome problems related to partially observable environments, state transition and connection cost models can be built to favour more recent observations using a finite-history-window (Kaelbling et al., 1996), and cached V_j values become stale using a *decay* model (Dorigo and Caro, 1999). In CRL, V_j information is decayed over time in the absence of new advertisements of V_j (see figure 4.4) values by a neighbour as well as after every recalculation of Q_i values. The absence of V_j advertisements amounts to negative feedback and allows the use of a cleanup updater to remove cache entries, actions and agents with stale values in the system. The rate of decay is configurable, with higher rates more appropriate for distributed systems with more dynamic network topologies.

4.2.6 The CRL Algorithm

The CRL algorithm can be used to solve system optimisation problems in a multi-agent system, where the system optimisation problem can be discretized into a set of DOPs, modelled as absorbing MDPs, in the following schema:

- A dynamic set of *agents* $\mathcal{N} = \{n_1, n_2, \ldots, n_M\}$, often corresponding to nodes in a distributed system.
- Each agent n_i has a dynamic set, $\mathcal{V}_i$, of *neighbours* where $\mathcal{V}_i \subset \mathcal{N}$ and $n_i \notin \mathcal{V}_i$.
- Each agent n_i has a fixed set of states $\mathcal{S}_i$, where $\mathcal{S}_i \subseteq \mathcal{S}$ and $\mathcal{S}$ is the system-wide set of states.
- Agents have both *internal* and *external states.*
  $Int : \mathcal{N} \rightarrow \mathcal{P}(\mathcal{S})$ is the function that maps from the set of agents to a non-empty set of internal

states that are not visible to neighbouring agents.

$Ext : \mathcal{N} \rightarrow \mathcal{P}(\mathcal{S})$ is the function that maps from the set of agents to a set of externally visible states. The relationship between internal and external states is the following:

$$\begin{array}{ll} Int(n_i) \subset \mathcal{S}_i & Int(n_i) \cup Ext(n_i) = \mathcal{S}_i \\ Ext(n_i) \subset \mathcal{S}_i & Int(n_i) \cap Ext(n_i) = \{\} \end{array} \quad \wedge \tag{4.1}$$

- We define a set of *connected states* between agents n_i and n_j as:

$$\mathcal{C}_{n_i n_j} = Int(n_i) \cap Ext(n_j) \quad where \quad n_j \in \mathcal{V}_i \tag{4.2}$$

$s \in \mathcal{C}_{n_i n_j}$ is a connected state where an internal state s at n_i corresponds to an external state s at n_j.

- Each agent n_i has a dynamic set of actions:

$$\mathcal{A}_i = \mathcal{A}_{d_i} \cup \mathcal{A}_{p_i} \cup \{discovery\} \tag{4.3}$$

where $\mathcal{A}_i \subseteq \mathcal{A}$, $\mathcal{A}_{d_i}$are the set of delegation actions, $\mathcal{A}_{p_i}$ are the set of DOP actions. The *discovery* action updates the set of neighbours, $\mathcal{V}_i$, for agent, n_i, and queries if discovered neighbouring agent n_j provides the capabilities to accept a delegated MDP from n_i. If it does, $\mathcal{A}_{d_i}$ is updated to include a new delegation action that can result in a state transition to $s \in \mathcal{C}_{n_i n_j}$ and the delegation of a MDP from n_i to n_j.

- There are a fixed set of state transition models, $P_i(s'|s, a)$, that model the probabilities of making a state transition from state s to state s' under action a.

- $D_i : \mathcal{S}_i \times \mathcal{A}_{d_i} \times \mathcal{S}_i \rightarrow \mathbb{R}$ is the connection cost function that observes the cost for the attempted use of a connection in a distributed system. $D_i(s'|s, a)$ is the connection cost model at agent n_i that describes the estimated cost of making a transition from state s to state s' under delegation action a.

- We define a cache at n_i as $Cache_i = \{(Q_i(s, a_j), r_j) : r_j \in \mathbb{R} \wedge a_j \in \mathcal{A}_{d_i}\}$. The value r_j in the pair $(Q_i(s, a_j), r_j)$ corresponds to the last advertised $V_j(s)$ received by agent n_i from agent n_j. For each $n_j \in \mathcal{V}_i$, $Cache_j$ is updated by a V_j *advertisement* for a connected state. The update replaces the r_j element of the pair $(Q_i(s, a_j), r_j)$ in $Cache_i$ with the newly advertised V_j value.

- $Decay(r_j) \rightarrow \mathbb{R}$ is the decay model that updates the r_j element in $Cache_i$,

$$Decay(r_j) = r_j.\rho^{td} \tag{4.4}$$

where td is the amount of time elapsed since the last received advertisement for r_j from agent n_j and ρ is a scaling factor that sets the rate of decay.

- A *cleanup updater* is available at each agent, n_i, to remove stale elements from its set of neighbours, $\mathcal{V}_i$, delegation actions, $\mathcal{A}_{d_i}$, connected states, $\mathcal{C}_{n_i n_j}$ and its $Cache_i$. When a $(Q_i(s, a_j), r_j)$ entry in the cache drops below a specified threshold, the cleanup updater removes the delegation action a_j from $\mathcal{A}_{d_i}$, the stale connected state s from $\mathcal{C}_{n_i n_j}$, and the pair $(Q_i(s, a_j), r_j)$ from

$Cache_i$. If after removing s, $\mathcal{C}_{n_i n_j} = \{\}$ for some neighbour n_j of n_i, then n_j is removed from $\mathcal{V}_i$.

- The distributed model-based reinforcement learning algorithm is:

$$Q_i(s,a) = R(s,a) + \sum_{s' \in S_i} P_i(s'|s,a).$$
$$(D_i(s'|s,a) + Decay\,(V_j(s'))) \tag{4.5}$$

where $a \in \mathcal{A}_d$. If $a \notin \mathcal{A}_d$, this defaults to the standard model-based reinforcement learning algorithm (Kaelbling et al., 1996) with no connection costs or decay function. $R(s,a)$ is the MDP termination cost, $P(s'|s,a)$ is the state transition model that computes the probability of the action a resulting in a state transition to state s', $D_i(s'|s,a)$ is the estimated connection cost and $V_j(s')$ is $r_j \in Cache_i$ if $a \in \mathcal{A}_d$, and $V_i(s')$ otherwise. Note that rewards that are received in the future are not discounted since agents do not learn about state transitions after successful delegation to neighbouring agent.

- The value function at agent n_i, V_i, can be calculated using the Bellman optimality equation (Bellman, 1957):

$$V_i(s) = \max_a [Q_i(s,a)]$$

4.2.7 Feedback, Convergence and Decentralised Coordination in CRL

Adaptation of system behaviour in CRL is a feedback process in which a change in the policy of any agent, or a change in the system's environment as well as the passing of time causes an update to the policy of one or more agents (see equation 4.5). In CRL, changes in an agent's environment provide feedback into the agent's state transition model and connection cost model, while changes in an agent's policy provides collaborative feedback to the cached V values of its neighbouring agents using advertisement. Time also provides (negative) feedback to an agent's cached V values using the decay model. As a result of the different feedback models in CRL, agents can utilise more information when learning a policy in a distributed system.

Agents can share information about their operation using collaborative feedback and their shared environment. This enables agents to learn from the behaviour of other agents and ultimately for agents to converge on similar policies. If agents converge on similar policies then, by virtue of the fact that agents with converged policies execute similar actions when they observe similar states in their environment, the behaviour of the agents can become coordinated in a shared environment.

4.2.8 CRL in the ACDL

K-Components provides support for an implementation of the CRL algorithm as a CRL policy in the ACDL. Similar to the RL policy in section 3.7.6, an agent is modelled as an adaptation contract and states are defined as predicates on component feedback states. Actions in CRL are different as there are three different types of actions: DOP actions, delegation actions and discovery actions. DOP actions and delegation actions can be implemented as component adaptation actions, while a discovery action

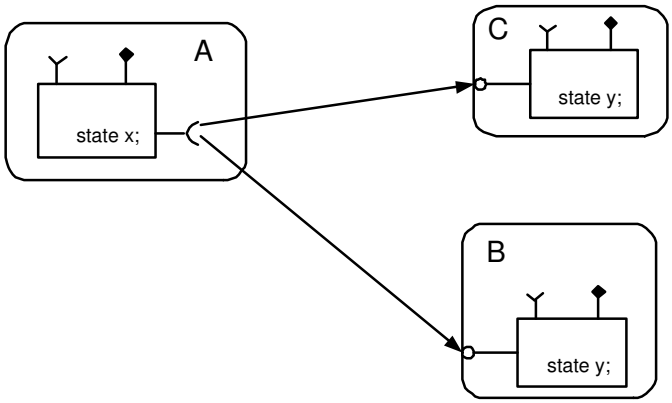


Figure 4.5: Connected states 'x' and 'y' between components A, B and C.

requires support for a discovery service to find new K-Components at runtime. Delegation actions use connectors defined on a component to forward the solution to a MDP to a connected component.

CRL also introduces several new abstractions to RL. These include connected states, a cache of V-values for connected states on neighbouring agents, a decay model for the values in the cache, an advertisement function, and a connection cost model. Connected states are established when one component binds to another and updates its local AMM to include the description of the target component. Component feedback states represent the connected states between the components. The connected states between components A, B and C in figure 4.5 are component feedback state x (an internal state at component A) and component feedback state y (an external state at components B and C). The AMM acts as the cache, $Cache_i$, of recently advertised values of component feedback states. The cached values of component feedback states can be decayed by specifying decay on a component feedback state in the ACDL. A decay definition contains the component feedback state, a scaling factor and the unit of time used to calculate the time difference. Also, the connection cost for a connected state can either be static or supplied by instrumented connectors as a status value that is available for monitoring as a feedback state on all connectors.

Advertisement functions can be implemented using RPC connectors, as well as specified in the ACDL using feedback events. In the ACDL, an advertisement feedback event can be defined as a component feedback state with an advertisement predicate that defines an advertisement period and a timeout (the current time left until the next advertisement), see table 3.4. A CRL policy has a default advertisement function based on RPC connector connectors if no advertisement feedback event is defined. In the ACDL, a CRL policy can be defined in an adaptation contract in the following format:

```
advertisement <ad_name>(<internal_state>,<external_state>,<seconds>);
decay <decay_name>(<state_name>, <scaling_factor>, <td>=1000ms);
crl_policy <rl_agent_name>(<agent_name>, <decay_name>, <ad_name>=NONE);
```

Definition 4.1: A collaborative reinforcement learning policy in runtime k is $CRL_i = \{RL_i, M_k, E_{remote_k}\}$, where

- $CRL_i \in Policy_i$ is the collaborative reinforcement learning policy that can be implemented in an adaptation contract ac_{ij}.

- $CRL_i \in \mathcal{CRL}$, where $\mathcal{CRL}$ defines the CRL system, i.e., the set of all CRL agents in the distributed system.
- M_k is the AMM that stores $cache_i$ of DOP connection costs from component c_i to connected components.
- E_{remote_k} is the set of remote feedback events that implement the advertisement function that specify the period for the asynchronous advertisement of V-values and also the set of feedback events that implement a decay function for a feedback state, defining the rate of degradation for the V-values cached in the AMM.

4.3 Summary

This chapter presented the CRL algorithm, a decentralised optimisation technique that addresses the requirement from section 2.3 for a coordination model that can establish and maintain system properties in a decentralised system.

The CRL algorithm extends RL with collaborative feedback and a decay model, and also defines different actions for coordinating the solution to distributed problems including delegation, DOP and discovery actions. The model of decay is useful for decentralised environments as it provides negative feedback on an agent's local, partial view of the system, requiring agents to constantly regenerate their view of the system. Advertisements update an agent's view of the system, providing collaborative feedback between agents. Collaborative feedback and coordination actions enable agents to coordinate their behaviour to optimise the solution to DOPs in a decentralised, multi-agent environment. Agents coordinate their behaviour so that the actions that each individual agent takes to solve a DOP produce near-optimal system behaviour. In chapter 6, it is shown how CRL can be used to build autonomic decentralised systems that adapt and optimise system behaviour to a changing environment using positive and negative feedback.

The next chapter describes the prototype implementation of the K-Component model.

Chapter 5

The K-Component Programming Model and Framework

"Idealism increases in direct proportion to one's distance from the problem"

John Galsworthy

This chapter describes a prototype implementation of the K-Component framework and programming model. The programming model and language mappings are described, and the main architectural components and their interfaces are discussed. Other implementation issues that are considered to be relevant to the prototype and the thesis are also covered.

This chapter is structured in a manner that roughly corresponds to the development lifecycle for a K-Component application. In section 5.2, the programming model for K-Components is described. This includes an overview of how to develop and deploy a K-Component, the language mappings for K-IDL and the ACDL, how to implement a component in section 5.3, incoming and outgoing connector implementations in section 5.4 and how to specify a component's adaptation logic in the ACDL in section 5.5. The implementation of a CRL policy in the ACDL is described, although an implementation of the actual algorithm is only presented later in chapter 6.

From section 5.6 until the end of the chapter, the services and data structures in the K-Component framework are discussed, including a description of the configuration manager, feedback event manager, AMM and adaptation contract manager. The main application programming interfaces used by component and adaptation contract developers are also described, including the configuration interface, the ArchReflect MOP and the ArchEvents interface. Other infrastructural services that are key to the framework are also described, including KOM and the configuration graph based on the extensible mark-up language (XML) document object model (DOM).

5.1 Overview of CORBA Mapping

Although the K-Component model is designed to be independent of the underlying distributed object computing platform, the implementation of the K-Component model, described in this chapter, has

been developed on a CORBA platform. CORBA is a distributed object computing middleware that is based on the client-server paradigm (OMG, Dec. 2002), and was chosen as an implementation platform due to the extensive research community working on reflective middleware architectures. This section describes the high-level mapping of K-Component concepts, such as the component model, connectors and the AMM to implementation structures built on top of CORBA's client-server model. The most notable difference between the development of CORBA programs and K-Components is that CORBA programs are generally either a client or a server program, while K-Components are developed as multi-threaded components that typically operate as both a client and server program. The main dependency between the current implementation and CORBA is that connectors and components use CORBA data-types (C++ mappings of IDL data-types) generated from the KIDL-compiler (see section 5.2.2).

The K-Component model extends CORBA's IDL-2 syntax, used to define interfaces to CORBA objects, as it does not support the specification of dependencies between CORBA objects[1]. The K-Component model provides K-IDL to specify component interfaces with explicit dependencies between components. K-Components also models dependencies between components at runtime using the AMM, while CORBA does not specify a model for managing runtime dependencies between CORBA objects. The AMM in K-Components is implemented in two parts: the configuration graph and the KOM registry. The configuration graph is implemented using the XML DOM (W3C, 1999) as a directed, configuration graph of components and connectors. The KOM registry stores the list of active components and connectors and provides the mapping from components and connectors in the configuration graph to the actual components and connectors in the runtime. KOM is a component model for C++ objects that was developed to support the AMM.

In CORBA, CORBA objects are typically incarnated as servants by registering the servant with the Portable Object Adapter (POA), and creating a globally unique identifier for the new object called the interoperable object reference (IOR). In K-Components, a component is identified using a `kref` identifier (see section 5.3.3), that contains an embedded IOR to locate the component. However, the component is not implemented as a servant, as components can be replaced at runtime and this would invalidate the published IOR. Instead, the component's incoming connector is incarnated as a servant, registered with the POA using an Object ID (generated by the POA) sand explicitly activated. The incoming connector is generated by the K-IDL compiler as a modified, templated Tie class (see section 5.4). The remote representation of a component using the incoming connector enables the replacement of component objects without the invalidation of the published IOR for a component, as the IOR to the servant remains valid even after the component is replaced. The incoming connector's POA-generated Object ID is also stored in the AMM, allowing it to be deregistered from the POA and K-Component runtime when necessary. The component itself is typically implemented as a KOM object. On the client-side, outgoing connectors use a CORBA stub to bind to remote components.

Finally, the K-Component runtime is similar to a CORBA server in that it can host many servants, although its implementation reflects the multi-threaded nature of K-Components. It is implemented

[1] Although IDL-3, part of the CORBA component model, supports the specification of explicit dependencies between components, its C++ mapping differs from K-IDL's mapping (OMG, 1999). Also the goals of the CORBA component model are different to those of the K-Component model. The CORBA component model is designed as a relatively heavyweight component model for enterprise computing, in contrast to the K-Component model's self-adaptive component model.

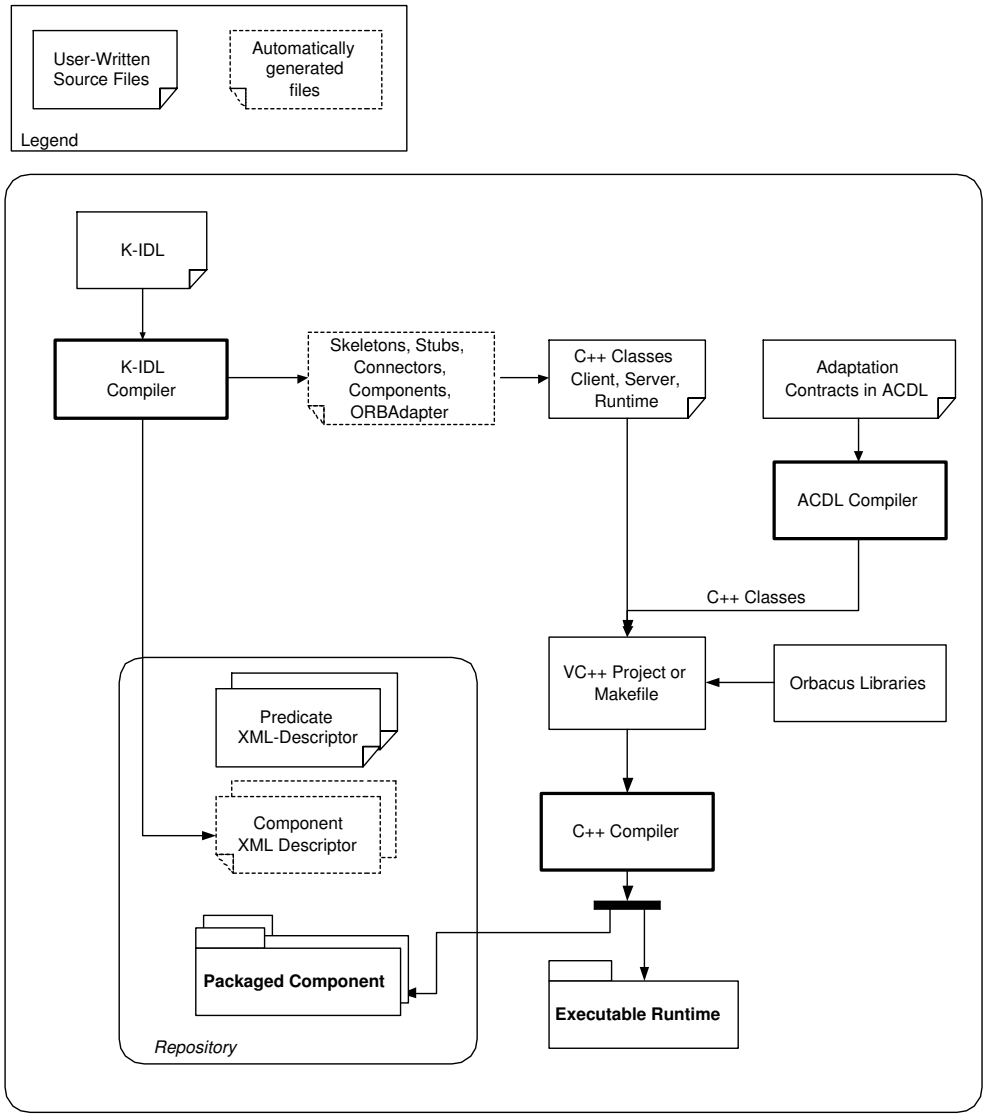


Figure 5.1: K-Component programming model.

as an active object that provides multiple threads for the different programs, including client threads, a CORBA server thread, adaptation contract threads and management threads.

5.2 Programming Model

K-Components provides a distributed programming model based on CORBA, that includes a K-IDL compiler for generating stubs, skeletons, component and connector implementation classes from both a component definition, the K-Component framework and the Orbacus/C++ libraries (Concepts, 2001). Orbacus is a CORBA-compliant ORB for C++ that is owned by IONA Technologies (Technologies, 2003). The objectives of the programming model include reducing the complexity of developing self-adaptive software and improving the maintainability of self-adaptive software. To help achieve these objectives, a component's public behaviour is specified and maintained separately from its adaptation logic. Component programming is based on K-IDL and C++, while adaptation logic programming uses the ACDL. As well as the compiler for K-IDL files, a compiler has been implemented for ACDL files that is used to generate adaptation contract classes.

Figure 5.1 illustrates the development lifecycle for a K-Component. The steps involved in specifying, implementing and deploying a K-Component are as follows:

1. Define a component interface in K-IDL and save as a .kidl file. A component developer specifies a component interface and any IDL constructs, such as IDL interfaces, structs and typedefs, required by the component in K-IDL.

2. The K-IDL compiler pre-processes and compiles the .kidl file to generate stubs, skeleton, connector and component classes in C++ (see figure 5.3). The pre-processor phase maps component interfaces into an extended version of IDL and the compilation phase then takes the extended IDL, compiles it and generates output classes in C++. The compiler also generates a component descriptor in XML that is stored in the Repository. The Repository is a local storage area for K-Component specific files.

3. The component programmer implements the component interface class to provide an implementation for the component.

4. The adaptation logic programmer specifies the server-side adaptation logic for the component in the ACDL. The ACDL compiler produces C++ adaptation contract classes as output.

5. The programmer writes a runtime, similar to a CORBA server (Henning and Vinoski, 1999), in which the component can be deployed. A project is defined as a makefile or a visual C++ project, includes the runtime, component and adaptation contract classes and is built as an executable.

The steps involved in specifying, implementing and deploying a client to a remote component are as follows:

1. The client programmer uses the client and proxy C++ classes produced by the K-IDL compiler to build two separate projects. The first project is a client component that uses a connector to bind to the component and use its services. The second project is the proxy for the remote component that encapsulates the CORBA stubs required to bind to the server component.

2. The adaptation logic programmer can specify client-side adaptation logic in the ACDL that is associated with the proxy for the remote component. The adaptation contract classes that are generated from the ACDL file are included in a project definition with the proxy component and built as a packaged shared library for Linux, or as a packaged dynamically linked library (DLL) for Windows32 platforms.

3. The programmer writes a runtime that includes the client component. A client application is built as a project that includes the runtime and client component. The packaged proxy component is loaded dynamically by the client when it binds to the server component. On binding, the adaptation contract in the proxy component is initialised and started. On unbinding, the contract is stopped and the library is unloaded.

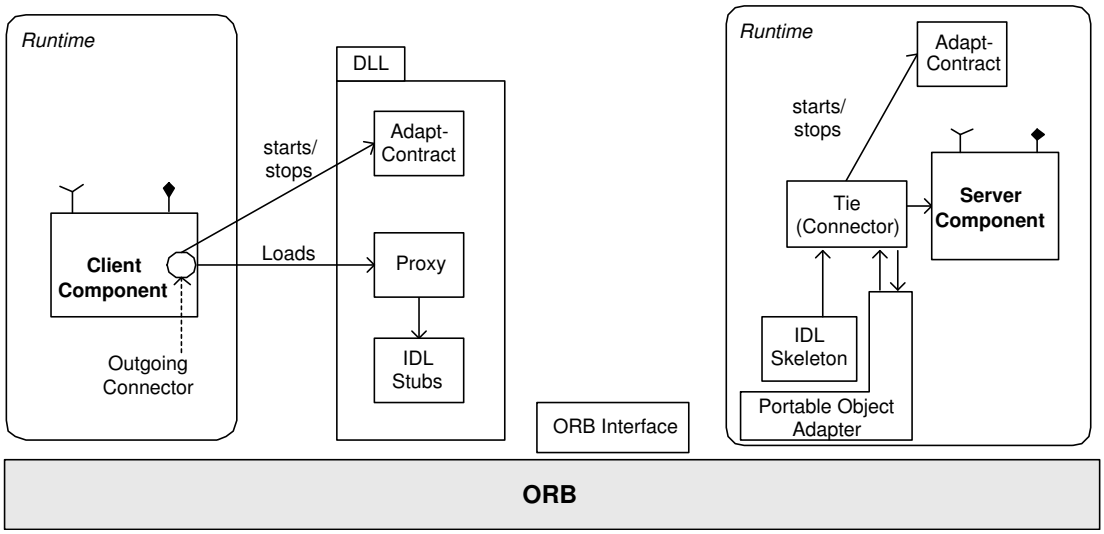


Figure 5.2: CORBA and the K-Component model.

5.2.1 Distributed System Architecture based on CORBA

The K-Component distributed system architecture is illustrated in figure 5.2. For purpose of clarity, the feedback event manager, configuration manager and the AMM have been omitted.

The server component is deployed in a K-Component runtime and can have an adaptation contract associated with it that is initialised by the component's incoming connector. The incoming connector is generated by the compiler as a modified, templated Tie class (see section 5.4). A Tie class is a C++ class template that is used to incarnate a servant (Henning and Vinoski, 1999), but is modified in K-Components to also play the role of the incoming connector.

The client component is also deployed in a runtime and it uses a dynamically loadable CORBA proxy component to communicate with the server component. The proxy component is loaded dynamically from a shared library in the Repository when the client attempts to bind to the server component. An adaptation contract, concerned with reasoning about connections to remote components, can be packaged along with the proxy in the shared library. The outgoing connector initialises the adaptation contract when it loads the proxy (see figure 5.29). As clients can have multiple outgoing connectors, client components may have multiple adaptation contracts concurrently reasoning about their operation and the operation of their outgoing connections. The shared library for the proxy is loaded dynamically using KOM (see section 5.3.2). When the client unbinds from the server, the associated adaptation contract is stopped and the shared library may be unloaded.

5.2.2 K-IDL Compiler

This section describes the mapping of K-IDL to IDL and then to C++. The mapping should address several requirements:

- The mapping should be clear and easy to understand.
- It should remain as close to CORBA as possible to maintain familiarity with CORBA programmers.

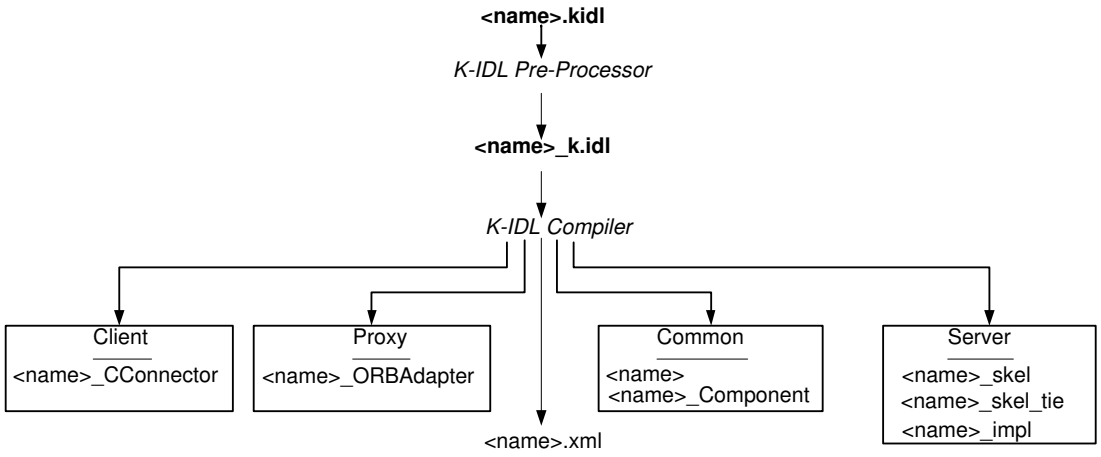


Figure 5.3: K-IDL compiler output.

- It should enforce as many constraints of the programming model as possible, e.g., restricting authorisation to invoke adaptation actions defined on components.

The K-IDL compiler compiles component definitions in K-IDL and generates C++ classes that can be used to implement a K-Component distributed application. The compiler has two phases, a pre-processor phase that maps K-IDL component definitions into an extended version of IDL, and a compilation phase that maps the extended IDL into C++ and generates component descriptors in XML. The K-IDL compiler is implemented as an extension to the Orbacus/C++ IDL compiler, version 4.1.1 (Concepts, 2001).

The compiler generates four distinct groups of C++ classes from a .kidl file (see figure 5.3): server-specific classes, client-specific classes, proxy-specific classes and finally classes that are common to the three other groups. The extensions to the Orbacus/C++ compiler include:

- generation of the empty component server implementation class `<component_name>_imp.h,.cpp` files.
- generation of the `<component_name>_skel_tie.h` file that includes a modified Tie template class that acts as the servant and class for incoming connector objects on a component server.
- generation of an outgoing connector class in `<component_name>_CConnector.h,.cpp` files.
- generation of the proxy class in the `<component_name>_ORBAdaptor.h,.cpp` files. The proxy class is packaged in a shared library.
- generation of a `<component_name>_Component.h,.cpp`, file that contains a component class that extends both the templated class `Component<>` from the K-Component framework and also the class `<component_name>KC`. The component class is extended by component server implementation classes, connector classes and proxy classes.

The compiler also generates a component descriptor in XML from a component interface specified in K-IDL. The component descriptor is used by the AMM to build the XML representation of the configuration graph of components and connectors (see figure 5.4). The configuration graph is represented using the AMM Document Object Model (W3C, 1999) (AMM-DOM), as it is implemented as

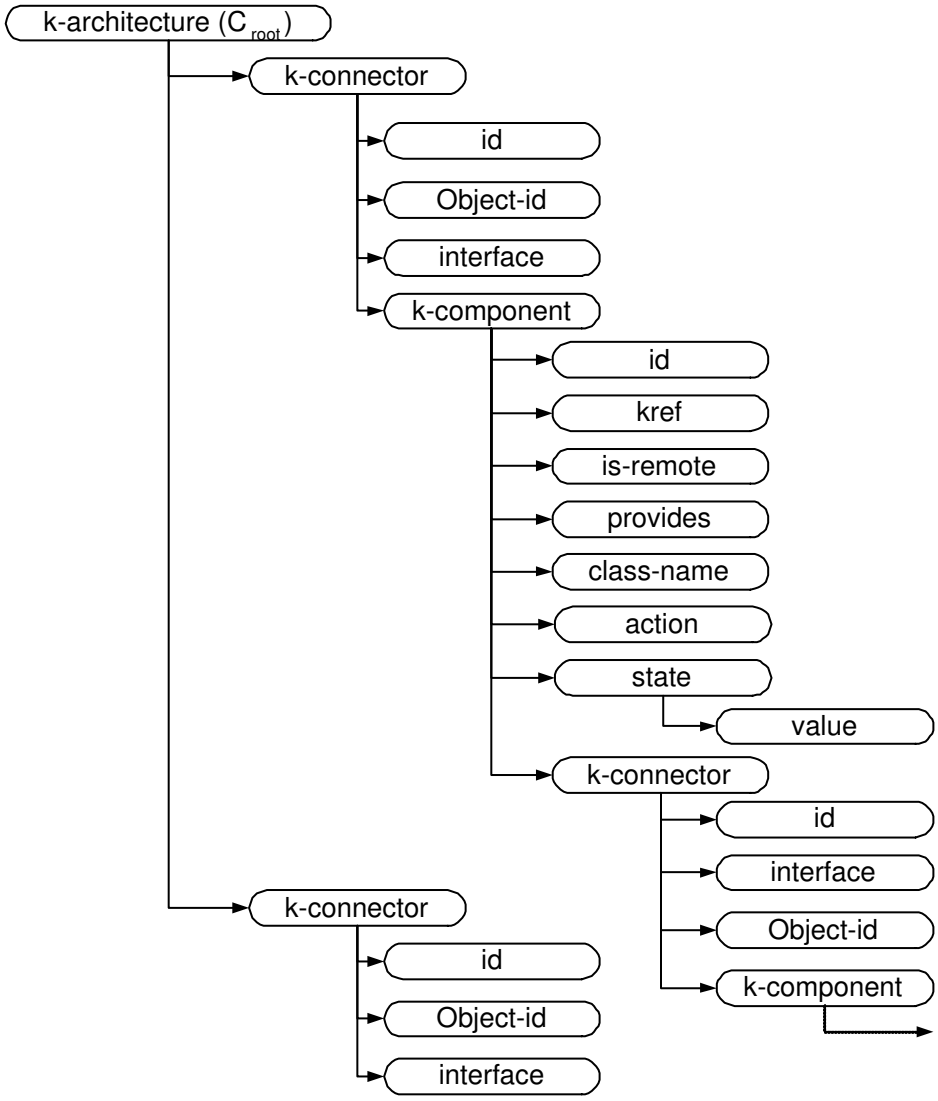


Figure 5.4: AMM-DOM configuration graph.

a DOM object using Apache's XML DOM Parser, Xerces-C++ (Project, Dec 2003). The component descriptor is stored in the Repository, from where it can be located and loaded at runtime. The configuration graph is automatically generated from component, connector and architecture descriptors at runtime (see section 5.7.1).

A component descriptor contains information required by the AMM to locate and operate on the component. The `id` attribute is a unique identifier used to find the component instance in the KOM-Registry. The `kref` attribute is also a unique identifier for the component and is used to locate the component. The `is-remote` attribute is used to determine whether or not the component is local or not. The `provides` attribute describes the interface provided by the component and `class-name` defines the class name that is used to implement the component. The component descriptor also contains the list of connectors, actions and feedback states defined on the K-IDL interface, as well as the cached values of the feedback states for remote components.

A connector is represented in the AMM-DOM by the `interface` it implements, the unique identifier, `id`, and an `Object-id` that is used by the configuration manager to shutdown an incoming connector if necessary

5.2.3 K-IDL to Extended IDL Mapping

K-IDL is mapped to an extended version of IDL by a pre-processor implemented using Lex, Yacc and C++. The pre-processor is part of the compiler, also implemented using Lex, Yacc and C++. A component interface is mapped to an IDL interface that inherits from the `KBind` interface and implements the provided interface. The `<component_name>` is modified by appending "KC" to it, allowing the modified IDL compiler to identify the interface as a component. An implementation for the operations in the `KBind` interface (see section 5.3.4) is generated by the compiler for both the component implementation class and the Tie class.

```
component <component_name> {
  provides <interface_name>;
};
```

is mapped to

```
interface <component_name>KC_impl : KBind, <interface_name> {
};
```

The mapping for a `uses` declaration in a component definition is problematic as IDL does not provide support for specifying required interfaces[2]. The mapping represents the only part of the K-IDL to IDL mapping that does not conform to IDL. Any `uses` declarations from K-IDL are left unchanged in the IDL mapping. The addition of the `uses` declaration required extending the Orbacus IDL compiler.

Feedback states defined on a component interface are mapped onto operations on the component's interface in IDL. The operations are prefixed with `k_state` so that the compiler can identify the state polling operations and generate the implementation to the method `poll_state` (see table 5.1)

[2]Although IDL-3 for the CORBA Component Model supports the specification of "uses" interfaces on components (OMG, 1999).

that takes as a string the name of the feedback state and delegates to the requested feedback state operation. For example:

```
state load;
```

is mapped to

```
double k_state_load();
```

Adaptation actions are also mapped onto operations on the component's interface in IDL. The operations are prefixed with `k_action` so that the compiler can identify the component adaptation actions and generate the implementation to the method `action` (see table 5.1) that takes as a string the name of the component adaptation action and delegates to the requested action operation. For example:

```
action store;
```

is mapped to

```
double k_action_store(in short strategy);
```

5.2.4 Extended IDL to C++ Mapping

The `uses` declaration is non-standard IDL and is mapped to a client-side connector object, encapsulated as a protected member variable in the component implementation class. Thus,

```
uses <interface_name> <connector_name>;
```

is mapped to C++ to the protected member variable

```
<interface_name>_CConnector* c_<connector_name>;
```

where `<interface_name>_CConnector*` is the outgoing connector class generated by the modified compiler and `c_<connector_name>` is a protected member variable in the `<component_name>KC_impl` class.

As in the standard IDL-to-C++ mapping (Henning and Vinoski, 1999), IDL interfaces are mapped to C++ classes. In K-Components, the component implementation class extends both the CORBA skeleton class and a templated component class in the K-Component framework, (see table 5.1). The templated component class extends the base class for all components and connectors, the `Object` class (see figure 5.5). The `Object` class provides common behaviours for connectors and components allowing them to be queried about the interface they provide, their class and their unique identity in the runtime. Namespaces are used to prevent conflicts with CORBA's base `Object` class.

The action operations in IDL have a non-standard C++ mapping. They are mapped to protected methods both in the base CORBA classes and the component implementation class. This hides component adaptation actions from 3^{rd} party components. Component adaptation actions can, however, be invoked by `ArchReflect` and `ConflictResolver` singletons, as they are both friend classes of the component implementation class.

An abridged version of the extended IDL to C++ mapping for the component implementation class is illustrated in table 5.1. Two methods in this class that merit discussion are `poll_state` and `action`. These methods are used by `ArchReflect` and `ConflictResolver` to invoke feedback state monitoring operations and adaptation actions. They are reflective operations that take the name of

```
class <component_name>KC_impl : virtual public POA_<component_name>KC,
                virtual public Component< <component_name>KC>
                virtual public PortableServer::RefCountServantBase {
   friend class ArchReflect;
   friend class ConflictResolver;
  protected:
   <interface_name>_CConnector* c_<connector_name>;
   CORBA::Double k_<state_name>;
  ...
  ...
  public:
  //  KBind Operations
  ...
  //  Reflective Operations
   CORBA::Double poll_state(const char* state_name) {
     ...
      if (!strcmp(<state_name>,state_name))
        return <state_name>();
     ...
   }
   CORBA::Double action(const char* action_name);
  ...
  //  Public IDL Operations
   virtual void buffer(const char* filename, BinaryFile& contents);
   virtual BinaryFile* retrieve(const char* filename);
  protected:
   CORBA::Double action(const char* action_name, CORBA::Short strategy);
   ...
   virtual CORBA::Double k_<state_name>();
   CORBA::Double k_<action_name>(CORBA::Short strategy);
};
```

Table 5.1: C++ Translation of Extended IDL.

the state or action as a parameter, resolve the name to a target method and delegate the request to the appropriate method, if one exists with the supplied name. The compiler generates the necessary code to perform the delegation:

Feedback States

Component programmers have to implement feedback state methods as thread-safe methods, so that different adaptation contract threads can safely invoke feedback state methods.

```
double k_<state_name>();
```

is mapped to

```
CORBA::Double k_<state_name>();
```

Adaptation Actions

Component programmers have to implement adaptation action methods as thread-safe methods, so that different adaptation contract threads can safely invoke adaptation action methods. Another constraint on adaptation action implementations is that they must complete in bounded time as they often lock structures shared with application-level components and connectors that must eventually

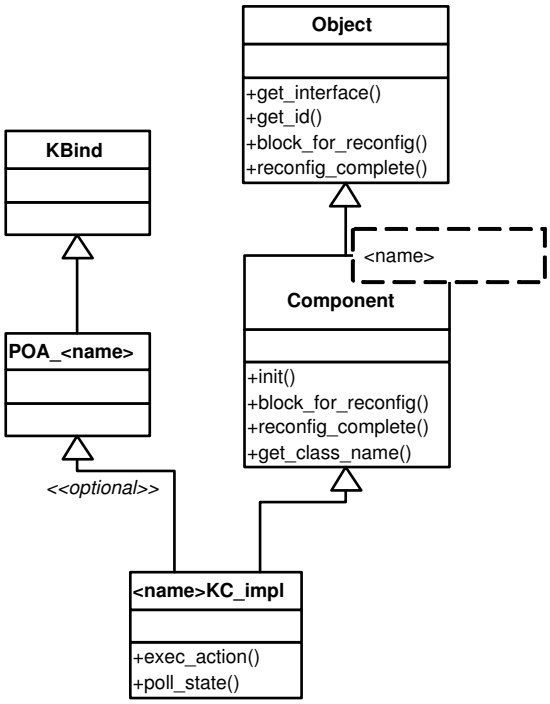


Figure 5.5: Class diagram of component.

be released.

```
double k_<action_name>(in short strategy);
```

is mapped to

```
CORBA::Double k_<action_name>(CORBA::Short strategy);
```

5.3 C++ Component

In K-Components, a component is any class that extends the templated class `Component<>` from the K-Component framework. Both CORBA objects, called server components, and proxy objects, called proxies, are components in K-Components. Components can be packaged as KOM objects (see section 5.3.2) so that they can be loaded from shared libraries at runtime. The class `Component<>` provides the following operations:

- `init(const char* name, <component_name>KC* ref)` used to initialise the component, e.g., in a proxy component it initialises the ORB.
- `void remove_component()` can be called by an incoming connector object to request that a component instance terminate its computation and communication with other components, and delete itself.

A CORBA server component is implemented as a class in C++ (see figure 5.5). The server component implementation class is generated by the K-IDL compiler and is called `<component_name>KC_impl`. The developer must provide an implementation for all pure virtual methods inherited from the CORBA skeleton class `POA_<component_name>KC`. The compiler generates the implementation for the infrastructural operations, such as those defined in `KBind` and `CORBA::Double poll_state(const char*)` and `CORBA::Double action(const char* action_name, CORBA::Short strategy)`.

5.3.1 Object Class

All component and connector objects are instances of the base `Object` class. The `Object` class provides operations common to both components and connectors, including `get_id`, `get_interface`, `block_for_reconfig` and `reconfig_complete` (see figure 5.5). Every connector and component object has a unique identifier. The unique identifier is used to store references to all connectors and components as `Object` instances in the KOM registry.

Both connector and component objects are subject to reconfiguration, and `block_for_reconfig` and `reconfig_complete` are used by the reconfiguration protocol implementation (see section 5.7.4).

5.3.2 KOM Objects

Components can be created as KOM objects. KOM is a platform-independent, non-distributed component model for C++ objects that was developed specifically for the K-Component model. KOM enables the creation and deletion of objects at runtime from dynamically linked libraries, external to the K-Component runtime's address space. KOM supports the creation/deletion of C++ objects from explicitly loaded DLLs in Win32 and shared libraries in Linux (see figure 5.6). It requires operating system support for the explicit linking of libraries of runtime, e.g., `LoadLibrary` in Win32 and `dlopen` in Linux, and the execution of functions at entry points to DLLs and shared libraries, i.e., `GetProcAddress` and `dlsym` for Win32 and Linux, respectively. It supports the transparent loading of shared libraries using the component naming scheme and the creation of KOM objects from the shared library using creator objects. As KOM provides the ability to load and unload objects at runtime using dynamically linked libraries, it must deal with authorisation and admission control for dynamically loadable code. KOM's solution is to ensure the code base is located inside a trusted local storage area, the Repository.

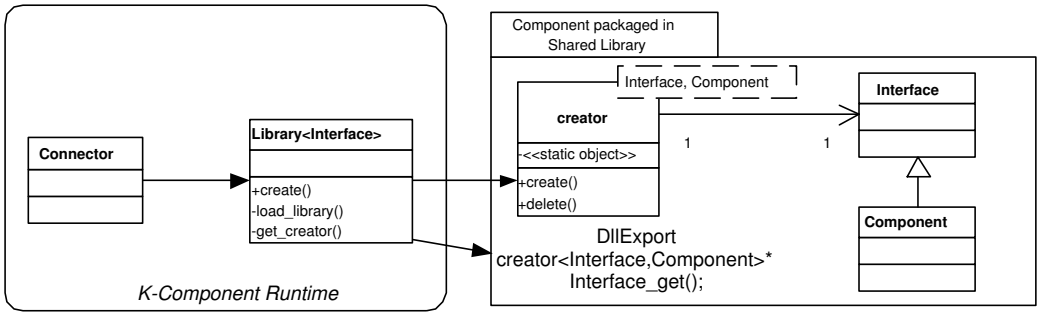


Figure 5.6: KOM creator registration.

Creator Objects

In Win32, memory allocated in a DLL cannot be deleted by a process external to that DLL. This means that objects created in a DLL cannot be deleted by any service resident in the runtime address space, as the DLL resides in a different address space. As a solution to this problem, a creator object is used to manage object creation/deletion from the DLL. The creator object is a static object that is created in the DLL when it is explicitly loaded and it is deleted when the DLL is unloaded (see figure 5.6). The creator class is a templated class and each creator object is responsible for the

creation/deletion of objects of a single type. Creator objects are registered with the KOM registry so that they can be used to create and delete objects in the DLL.

5.3.3 Component Naming Scheme

Connectors use a component reference called a `kref` to bind to components. In K-Components, a `kref` can denote a component either a local KOM object or a remote CORBA object, and have the following format:

```
protocol://RemoteAddr//LocalAddr//Interface/ClassName/CompID
```

where

- `protocol` is either `LIB` or `IOR` indicating that the component is either a local KOM object or a remote CORBA object.
- `RemoteAddr` is the stringified Interoperable Object Reference (IOR) (Henning and Vinoski, 1999) if the component provides the implementation for a CORBA object and empty if the component is a non-distributed object in a local library.
- `LocalAddr` is the name of the shared library that represents either a KOM component or the proxy to the remote component. As the shared library should be deployed in the Repository, clients can resolve its location using only the library name.
- `Interface` is the name of the interface provided by the component.
- `ClassName` is the name of the component class that implements the provided `Interface`.
- `CompID` is a unique identifier for the component generated by the runtime.

A `kref` is generated by an incoming connector in the current prototype and is distributed to clients in text files, (see table 5.3). The `kref` serves two main purposes in K-Components. Firstly, it is used by the KOM registry to identify component instances in the runtime, so that they can be introspected and adapted. Secondly, it is used as an object reference to bind to remote objects or load KOM objects if the component is not already loaded in the runtime. If the component is a KOM object, an instance can be created by loading the local library from LocalAddr. If the component is remote, the IOR embedded in the `kref` can be used to bind to the CORBA component. IORs refer to exactly one object instance, are strongly typed, can be persistent and can be nil. One performance problem with embedding stringified IORs in a `kref` is that they typically vary in length from 200 to 800 bytes, although there is no upper limit on their size (Henning and Vinoski, 1999).

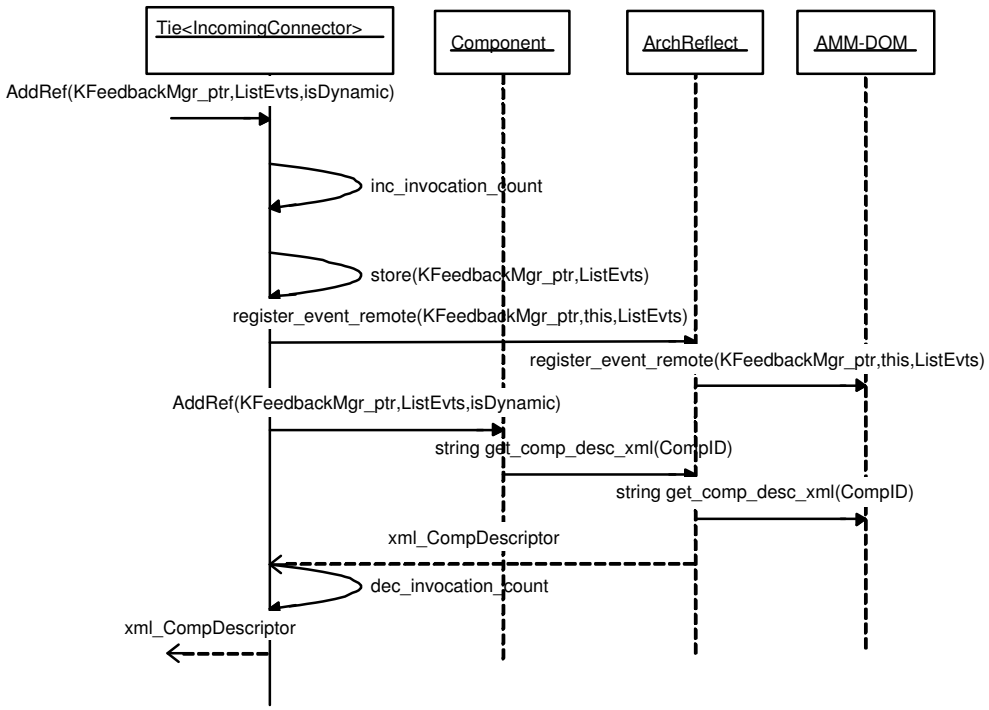


Figure 5.7: Server-side AddRef.

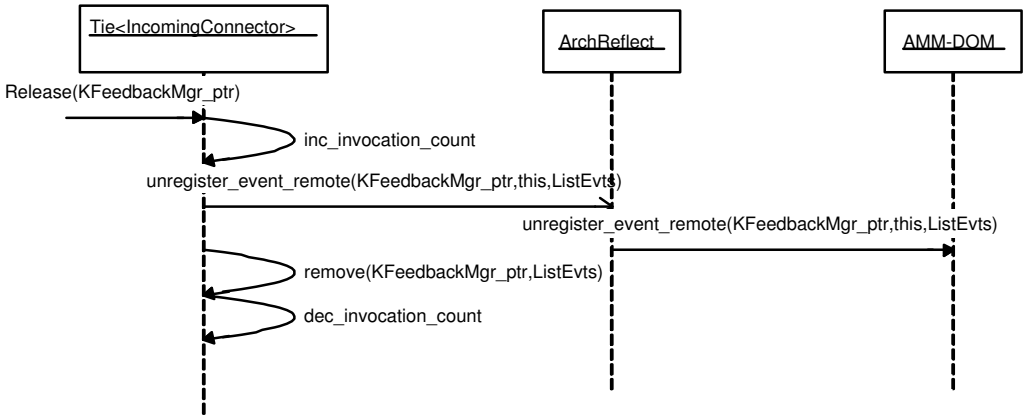


Figure 5.8: Server-side Release.

5.3.4 KBind Interface

The `KBind` interface defines a set of behaviours provided by every CORBA component and is described in table 5.2. It allows the registration of feedback events by clients on servers and the transfer of component descriptions from servers to clients. The operations supported include a registration service, `AddRef()`, that allows clients to register feedback events and a reference to the feedback manager CORBA object with the server and receive a stringified component descriptor as a return value. `Release()` allows a client to deregister its feedback events and its reference to the feedback manager with the component server.

```
// KBind Interface is implemented by every Component
KBind {
   string k_meta_AddRef(
       in KFeedbackMgr kref, in KEventRegistrationList list_events,
       in boolean isDynamic) raises (AlreadyRegistered);
   short k_meta_Release(in KFeedbackMgr kref)
       raises (NotRemovable, NotRegistered);
    ReConnection k_meta_heartbeat();
};
```

Table 5.2: KBind Interface provided by every component.

The order in which the `KBind` operations are invoked when a client's outgoing connector binds to a server is illustrated in figure 5.7, and unbinding from a component server in figure 5.8. It follows the sequence:

1. `bind()`: the client attempts to bind to the remote component. Before returning control to the client, the `AddRef()` operation is invoked on the server.
2. `AddRef()`: the registers its set of feedback events with the remote component's AMM as well as a reference to its feedback event manager. The `AddRef` method does not maintain a reference count of the number of connected clients.
3. `unbind()`: the client attempts to unbind from the remote component. Before returning control to the client, `Release()` operation is invoked on the server.
4. `Release()`: the client is deregistered, removing its feedback event manager reference and any registered feedback events.

The `heartbeat()` operation allows the Connection Manager (see figure 5.36) to ping a server to see if it is still available. This is useful in decentralised environments where connections are frequently dropped due to host unavailability or migration. If a `heartbeat()` fails a user defined number of times, the Connection Manager sets the value of the `status` feedback state in a connector to be an error. In the prototype, the code that implements the `KBind` operations in the incoming and outgoing connector class as well as the component implementation class is generated by the compiler.

5.3.5 Component Creation and Deletion

Components can be created as either normal objects, e.g., instantiated as CORBA objects, or using KOM (see section 5.3.2). Components are identified using a `kref` (see section 5.4.3). Similarly components can be deleted either as normal C++ objects or using KOM. When components are created and deleted the AMM has to be updated, including the configuration graph, AMM-DOM, and the KOM registry (see figures 5.9, 5.10). When KOM objects are created, they are automatically added to the AMM, while ordinary C++ objects require explicit registration/deregistration using the `ArchEvents` interface. CORBA objects also have to be registered with the AMM along with a reference to its ObjectID to enable the configuration manager to shutdown the object if necessary.

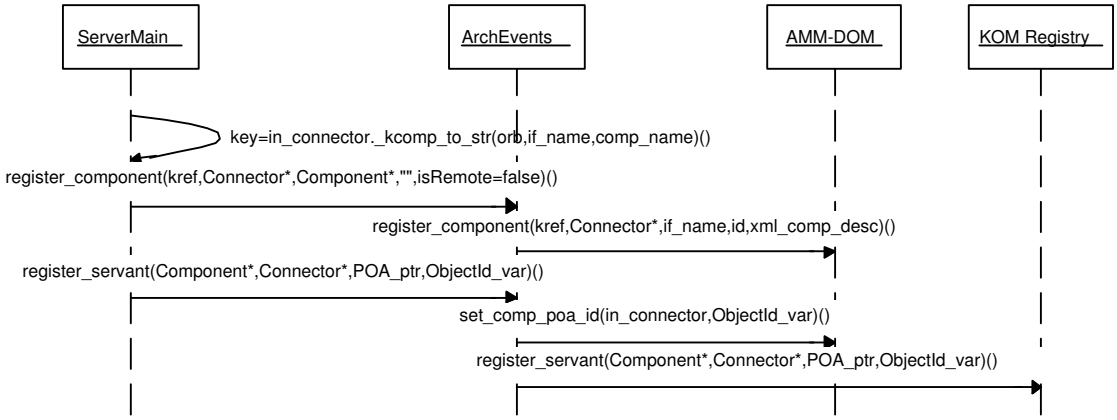


Figure 5.9: Component creation and registration.

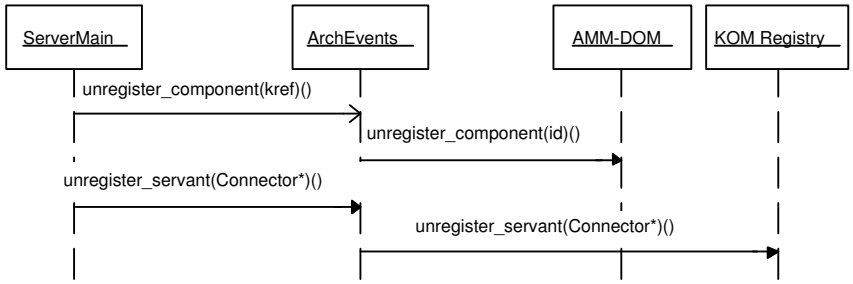


Figure 5.10: Component deletion and deregistration.

Generally components are created as KOM objects, rather than CORBA objects, as they are platform independent and easily repluggable.

5.3.6 Component Runtime

Component programmers have to write a runtime in which to deploy component server objects. A sample, edited runtime can be found in table 5.3. A runtime is a C++ application that contains an application thread, `ServerMain` in the above example, that is started by the configuration manager after calling `configuration_manager::init`. `ServerMain` also acts as the root component for the AMM.

In the `ServerMain::run` method, there are several changes in progamming style to standard CORBA. The incoming connector acts as the servant and it must be explicitly activated by a POA, so that it can be shutdown if necessary by the configuration manager (see section 5.6.1).

The incoming connector (or servant) also provides a method, `_kcomponent_to_string`, to create a component reference, `kref`. The method uses the `object_to_string` method on the ORB interface to create a stringified IOR that is inserted in the `kref`. It also uses the component's class name and the proposed name of the proxy's shared library, used by KOM, to generate a `kref`.

```
...
class ServerMain : public Main
{
public:
  int initialise(int argc, char* argv[]) {
              // ORB initialisation here
  }
  virtual void run() {
    ...
    COMPONENT_NAMEKC_impl* comp = new COMPONENT_NAMEKC_impl(rootPOA);
    POA_COMPONENT_NAMEKC_tie<COMPONENT_NAMEKC_impl> in_connector(comp);
    ...
    // Store ObjectID from POA's active object.
    // Explicit activation of servant.
    PortableServer::ObjectId_var oid =
      PortableServer::string_to_ObjectId("COMPONENT_NAME");
        cm_poa->activate_object_with_id(oid, &in_connector);
     ...
    CORBA::String_var kref =
    in_connector.
   _kcomponent_to_string(orb, "COMPONENT_NAME_impl", "COMPONENT_NAMELibrary");
    configuration_manager::register_connector(&in_connector, 0, "files");
    configuration_manager::register_component(kref.in(),
                        &in_connector, comp, "", false, false);
   configuration_manager::register_servant(comp, &in_connector, cm_poa, oid);
    ...
    orb -> run();
    configuration_manager::deregister_servant(comp);
  }
private:
   CORBA::ORB_ptr orb;
   int argc_;
   char** argv_;
};
int main(int argc, char* argv[]) {
   JTCInitialize initialize;
   ServerMain* sm = new ServerMain;
   sm->initialise(argc, argv);
   int res = configuration_manager::init(argc, argv, sm);
   return configuration_manager::shutdown();
}
```

Table 5.3: C++ runtime with a deployed component.

5.4 Incoming and Outgoing C++ Connectors

The compiler generates both incoming an outgoing connector objects from component definitions in K-IDL. Incoming connector objects are instances of a modified Tie class, while C++ outgoing connector objects are instances of a K-Component specific connector class.

The outgoing connector class `<component_name>KC_CConnector` is generated from a uses interface definition in extended IDL. Instances of this class are outgoing connector objects. The class conforms to the interface to `<component_name>KC_Component` by extending it, but it overrides its methods to provide connector specific behaviours. The outgoing connector objects use a dynamically loadable proxy component that provides CORBA IIOP stubs as transport. The proxy component is loaded

from the Repository on connector binding using KOM. Class diagrams for the outgoing and incoming connectors are shown in figure 5.11 and figure 5.12, respectively.

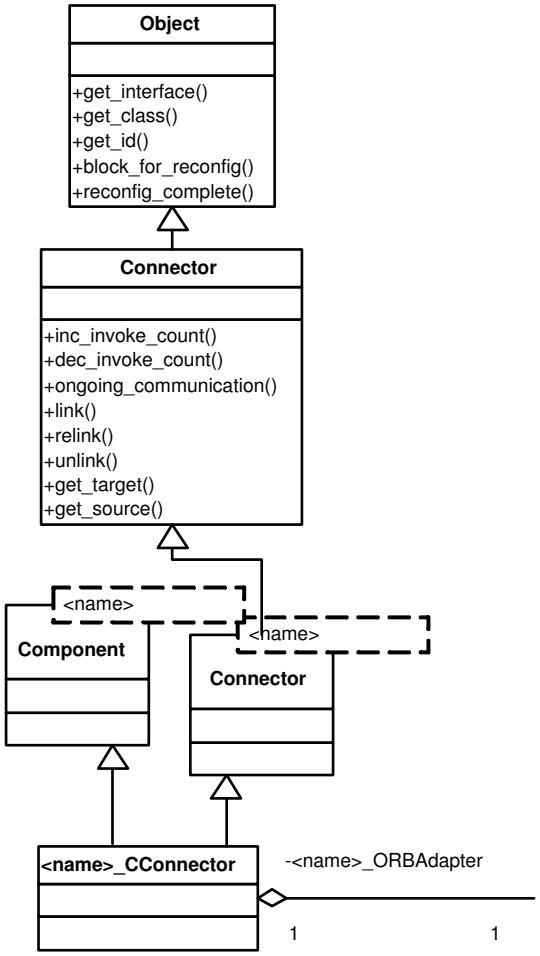


Figure 5.11: Class diagram of the outgoing connector.

5.4.1 Tie Class as an Incoming Connector and a CORBA Servant

A servant class is used to create and incarnate CORBA objects that can be accessed by remote clients (Henning and Vinoski, 1999). In Orbacus, there are two possible servant classes that can be used to incarnate CORBA objects, the `<name>_impl` class and a Tie class `<name>_tie`. The K-IDL compiler generates both of these classes, but the Tie class is used to incarnate the CORBA object. The Tie class generated was modified from the Orbacus compiler to be an incoming connector object (see figure 5.12):

```
template<class T> class POA_<name>_tie : virtual public POA_<name>KC,
    virtual public Connector, virtual public JTCMonitor {
   ...
};
```

A Tie class is a C++ template that is used to instantiate a concrete servant. The implementation of all methods in the Tie servant is delegated to the component instance, (called the tied object in

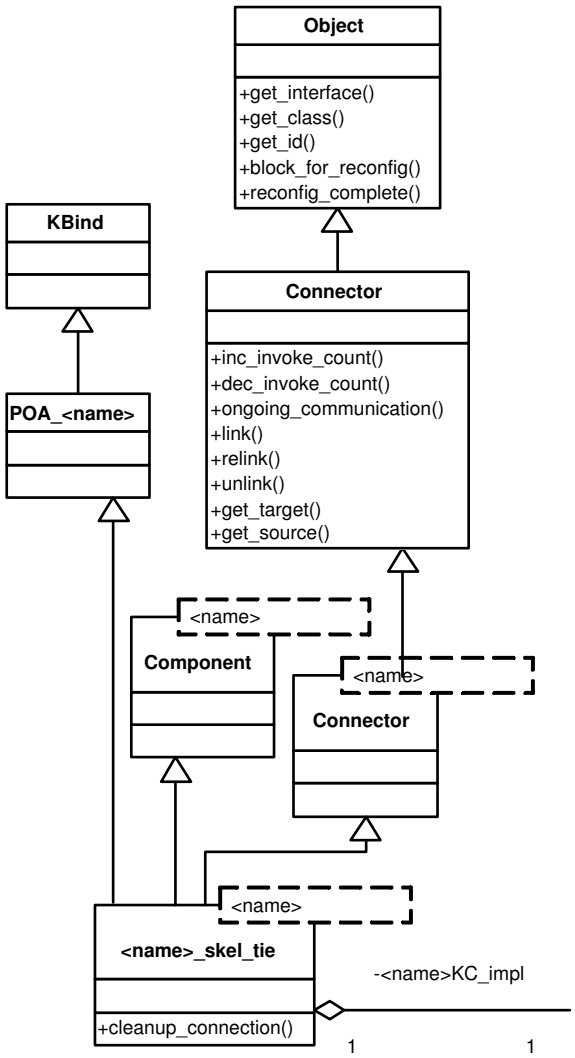


Figure 5.12: Class diagram of an incoming connector.

CORBA terminology). The Tie class implements the Object Adapter pattern (Gamma et al., 1995) where the Tie class plays the role of the adapter object and the component instance is the adaptee object. Incarnating a servant involves creating a new CORBA object, registering the servant with the Root POA (Henning and Vinoski, 1999), and creating a reference for the new object.

5.4.2 Connector Creation and Deletion

The connector constructor requires that programmers supply an identifier for the connector, a reference to its encapsulating component and an optional reference to its target component (see figure 5.14 and figure 5.15):

```
<component_name>_CConnector* c
= new <component_name>_CConnector(const char* id, Object* src, const char* kref);
```

The base `Object` class for the creator stores stringified names for the `uses` interface, the connector class and its `id`. A connector object is identified using the fully qualified `connector_id`:

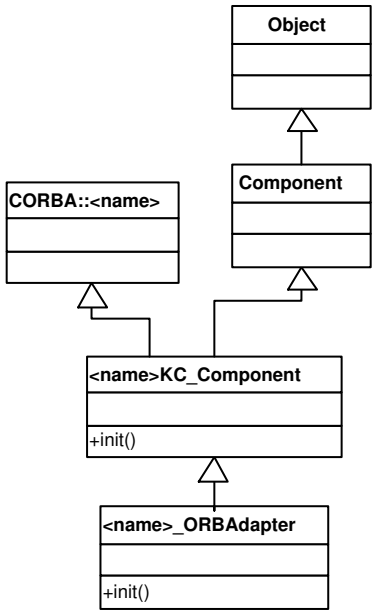


Figure 5.13: Class diagram of a client-side CORBA proxy.

```
<Component>::<Interface>::<connector_id>
```

This identifier is used in the ACDL to acquire a reference to the connector object. When connectors are deleted, they are also deregistered from the AMM (see figure 5.16 and figure 5.17).

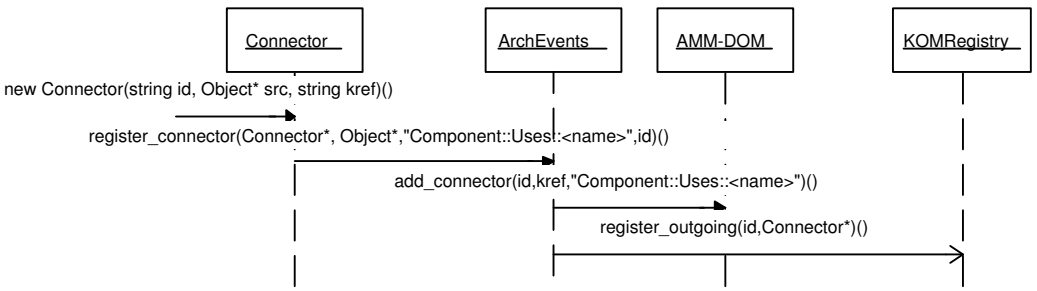


Figure 5.14: Outgoing connector creation and registration.

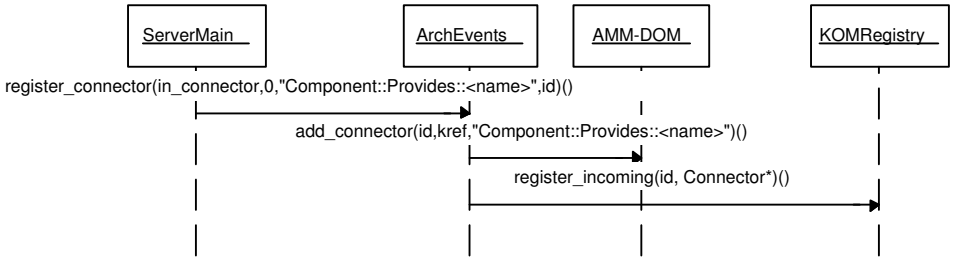


Figure 5.15: Incoming connector creation and registration.

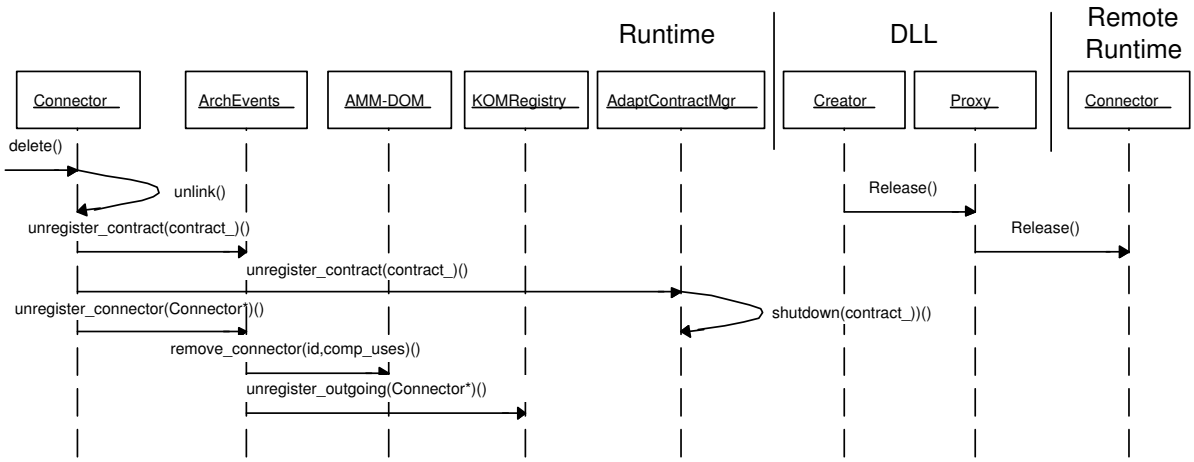


Figure 5.16: Outgoing connector deletion.

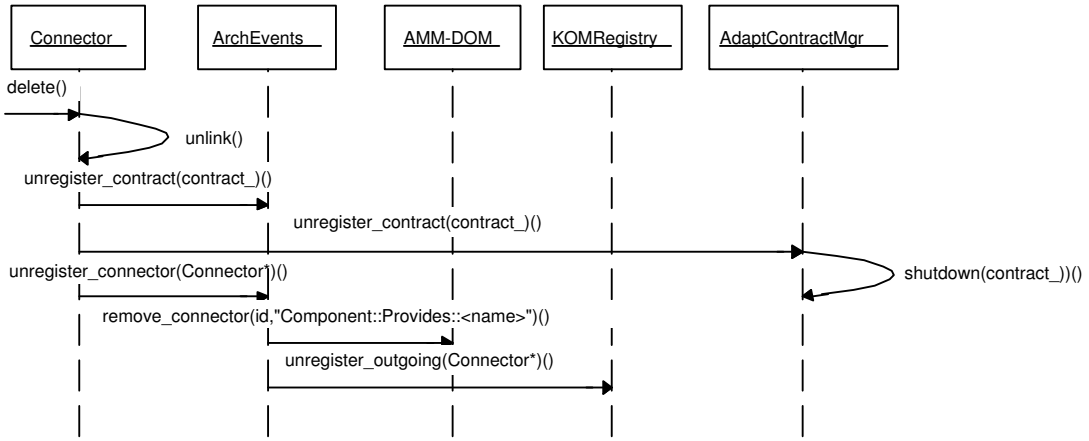


Figure 5.17: Incoming connector deletion.

5.4.3 Connector Binding

Connectors can bind to a remote component using either the connector constructor or the `bind(const char* kref)` method defined on all connectors. Connectors can unbind from a remote component using the `unbind()` method. Connectors can be bound to components by adaptation contracts that call the bind operation defined on the `ArchReflect` MOP and is used to acquire a reference to the connector from the KOM registry and `kref` is the component reference:

```
double ArchReflect::bind_connector(const char* connector_id, const char* kref);
```

5.4.4 Exception Handling in K-Components

There are three different types of exceptions that can typically be thrown in a K-Component application:

- ANSII C++ exceptions
- CORBA/C++ exceptions
- K-Component exceptions

A hierarchy of K-Component-specific exception classes are defined in the K-Component framework. The framework is designed to provide a uniform exception handling framework to the programmer. To this end, CORBA exceptions that are thrown in component proxies are wrapped in K-Component exceptions before being propagated to the application level. This enables programmers to write the same error-handling client code for components that reside in both local libraries and remote CORBA components. Some of the base exception classes in the hierarchy include `KComponentReferenceException`, `KConnectorException`, `KAMMException` and `KRegistrationException`. All these exceptions have many subtypes, e.g.,
`KComponentReferenceException` has subtypes `KHostNotFound` and `KLibraryNotFound`.

5.4.5 Comments on the Programming Model

The runtime provides thread management services and application programs are encapsulated in a managed thread object to conform to the JTC threading management model (see section 5.6.1). K-Component application programmers are restricted to the creation of threads that conform to the threading model.

The use of templates in the design of component and connector classes enables the framework to perform runtime type checking on KOM objects loaded from libraries, using the `dynamic_cast` operator (Stroustrup, 2000). This helps meet the goal of maintaining system integrity by ensuring that KOM objects created at runtime are the expected type.

5.5 Adaptation Contracts in the ACDL

When a component programmer has finished implementing a component, a client component and runtimes for both, an adaptation logic programmer can specify a system's adaptive behaviour on both the client and server side using the declarative ACDL. This section explains how ACDL contracts, types, operators, expressions, adaptation actions, monitoring operations and policies are mapped to C++ constructs by the ACDL compiler. The mapping should address several requirements:

- The ACDL should remain as close to C++ (without the pointers) as possible to maintain familiarity with C++ programmers.
- The mapping should enforce as many constraints of the programming model as possible at compile time, e.g., flag type-incompatible reconfiguration requests as errors at compile time.
- The generated C++ should not produce any compilation errors and runtime errors should be handled by exceptions and not result in serious system errors, such as program failure.

5.5.1 ACDL to C++

The ACDL is mapped to C++ using a compiler written based on the tool ANTLR version 2.4 for C++ (ANTLR, 2003). The following section introduces the mapping for ACDL to C++ code that can then be included in a component project and built. Firstly, the simplest mappings for the ACDL are the declaration of identifiers, operators and primitive types. Identifiers, primitive types and operators are mapped unchanged to the generated C++ code.

```
...
namespace KC {
class public contract : public Object, public JTCMonitor, public JTCThread {
public:
    contract();
    ~contract();
    virtual void validate();
    virtual int init(Connector* c,
       const char* dll_name, int sampling_time_interval);
    virtual int shutdown();
    virtual void run();
};
extern "C" {
        ACDL_API creator_base<contract>* contract_get();
}
}
```

Table 5.4: Header file for an adaptation contract in C++.

Adaptation Contracts and Handlers

The two main modular units of ACDL code are adaptation contracts and handlers. They are mapped to a class that extends the framework's `contract` class and a function, respectively. Instances of subclasses of the `contract` class are autonomous objects that run in their own thread of control and are managed by the adaptation contract manager. In ACDL to C++ mappings presented in this section, error checking code has been omitted for purpose of improving code readability. A simple contract

```
[incoming | outgoing] <contract_name> (<connector_id>){
   [statement]*
};
```

maps to a C++ class that extends the K-Component framework class, `contract`:

```
class contract_<contract_name> : public contract {
   ...
};
```

The base `contract` class for all generated adaptation contract classes is summarised in table 5.4. The method `init` is called by an incoming or outgoing connector when the contract is loaded. It is used to provide a reference to the associated connector object, register all feedback events declared in

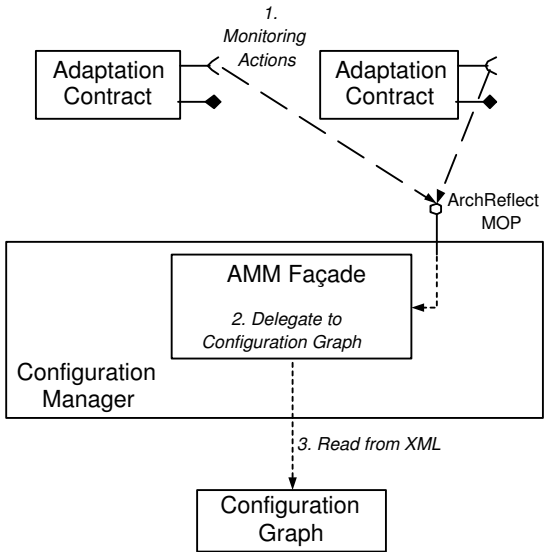


Figure 5.18: Monitoring from a contract.

the contract, and load all the predicate descriptors into predicate member variables. The `validate` method contains other adaptation logic declared in the contract, such as `if-then` adaptation action rules and is called by the `run` method. The `shutdown` method is called to deregister the contract with the adaptation contract manager and force the thread to exit. Finally, the `contract_get` function is exported from the DLL by defining `ACDL_API` as `_declspec(dllexport)` in the header file. This allows objects in the runtime create instances of the `contract_storage` class at runtime without knowledge of the `contract_storage` type. The `contract_get` function returns a `creator` object that is used to create and delete instances of the `contract_storage` object at runtime (see figure 5.7.2).

A handler maps to a function definition that is referenced as a function pointer by `configuration_manager::register_event_local(...)`. Thus,

```
handler <handler_name>() {
}
```

maps to:

```
void callback_<handler_name>(double val, const char* params)
{
}
```

A reference to a handler function can be stored as a C++ function pointer with a locally registered feedback event, as part of an event-condition-action definition, and invoked when the feedback event is raised.

States and Actions

Adaptation contracts can monitor component feedback states by calling `poll_state` that returns the value of the component feedback state cached in the AMM (see figure 5.18).

```
double poll_state(<comp_id>, <state_name>);
```

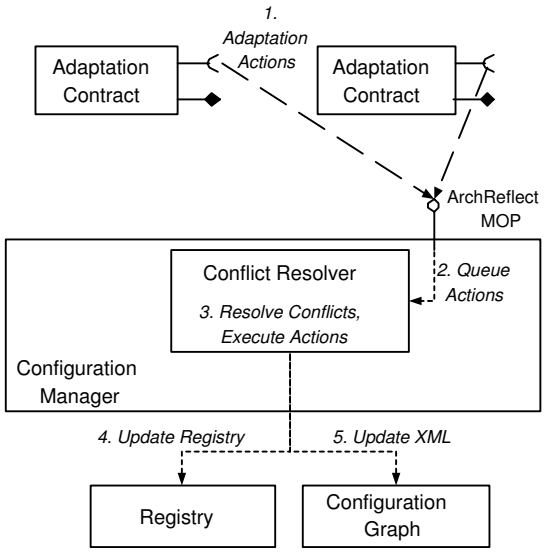


Figure 5.19: Architectural adaptation action execution from a contract.

maps to

```
double ArchReflect::poll_state(<comp_id>, <state_name>);
```

Component adaptation actions and architectural adaptation actions can also be executed from adaptation contracts.

There are two ways to execute component adaptation actions. Either the `ConflictResolver` can queue requests to execute adaptation actions from adaptation contracts (see figure 5.20), and execute them after checking for conflicts, or adaptation actions can be executed directly on components allowing contracts to receive a reward from the components for executing the action. Architectural adaptation actions are performed directly on the AMM (see figure 5.19).

Both architectural and component adaptation actions are assumed to finish in bounded time. An ACDL action :

```
double action(<action_id>[, Priority[, strategy]]);
```

maps to

```
double ArchReflect::action(const char* action_id,
    Priority=None, CORBA::Short strategy);
```

Feedback Events

Feedback events and predicates can also be defined in the ACDL as follows:

```
predicate <pred_comp_state>(<XML_file>);
event <event_name>(<state_name>,<pred_comp_state>,Priority,<handler_name>);
```

A predicate definition maps to, in effect, a reference to an XML file containing the predicate. In the ACDL mapping, it is mapped to a member variable of type `string` in the contract:

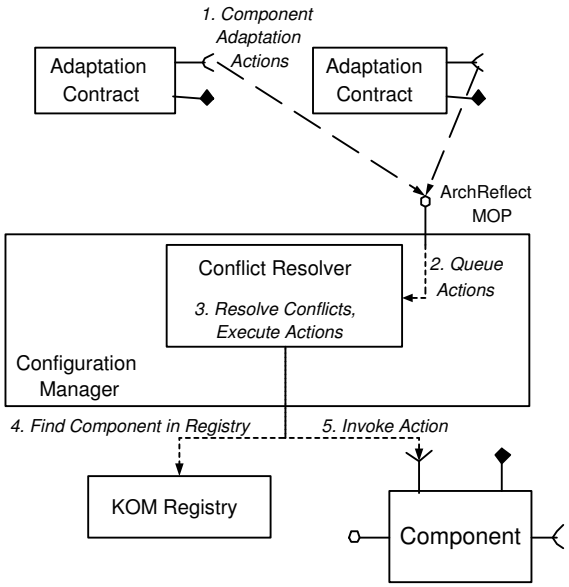


Figure 5.20: Component adaptation action execution with conflict resolution.

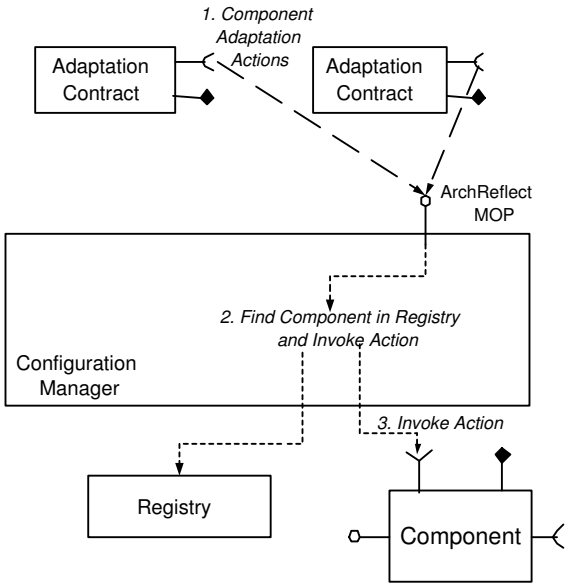


Figure 5.21: Component adaptation action execution with a reward model (and no conflict resolution).

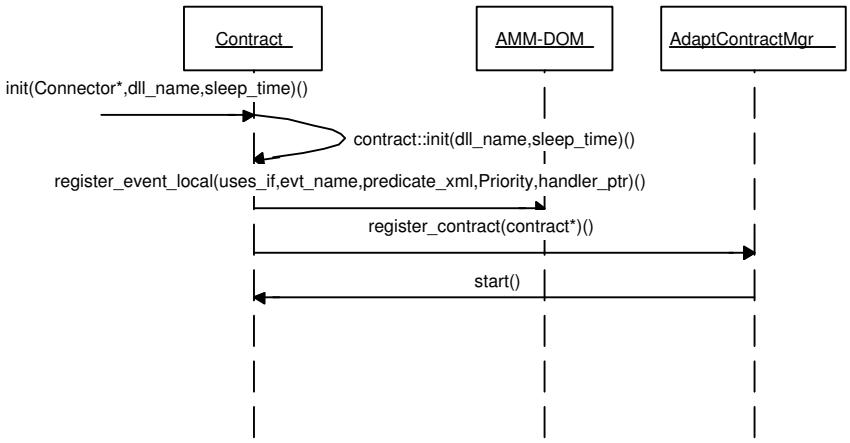


Figure 5.22: Adaptation contract initialisation.

```
string pred_comp_state;
std::ifstream f_pred_comp_state Storage(”<XML_file>”);
f_pred_comp_state >> pred_comp_state;
```

and it is initialised in the `init` method of the contract in C++ (see figure 5.22) along with the feedback event. The method `init` is called when the contract is started and it registers the feedback event with the AMM. When a client attempts to bind a connector to a remote component, the connector checks the AMM to see if any feedback events have been registered for it, and if they have, they are then registered with the remote component using the `KBind` interface `AddRef` operation. The `AddRef` operation is also used to register a reference to the feedback event manager for the K-Component.

```
virtual void init(..) {
...
   std::ifstream xml_pred_com_state("F:\\Repository\\ref\\<XML_file>.xml");
   xml_pred_com_state >> pred_com_state;
   configuration_manager::register_event_local(connector_->get_interface(),
    <state_name>,pred_com_state.c_str(), Priority, <fn_ptr_handler_name> );
...
}
```

States, Connectors and Components

The definitions of state, connectors and components:

```
state <state_name>(<component_id>);
```

maps to:

```
string <state_name> = resolve_kref::get_class_name(<component_id>);
```

```
<state_name>.append("::");
<state_name>.append("<state_name>");
```

A connector definition is mapped to a stringified `connector_id` from section 5.4.3. The string is used to identify the connector object uniquely in `ArchReflect` operations.

```
connector <connector_name>(<connector_id>);
```

maps to:

```
string <connector_name>("<connector_id>");
```

A component definition is mapped to a stringified representation of either its `kref` or a `component_id`. So,

```
component <component_name>(<kref>);
```

maps to (for `kref`):

```
std::ifstream ior_file_component_name
    Storage("F:\\Repository\\ref\\<component_id>.kref");
string component_name;
ior_file_component_name >> component_name;
```

and (for `component_id`)

```
component <component_name>(<component_id>);
```

maps to:

```
string <component_name>("<component_id>");
```

Architectural Adaptation Actions and Jitter

There are two architectural adaptation actions defined. The first, `rebind_connector`, unbinds an existing connector from its target component and then binds it to a different target component. It uses the connector identifier and `kref` to identify the connector and the component in ArchReflect. ArchReflect subsequently uses the KOM registry to acquire a reference to the connector object and then calls the `unbind` and `bind` operations on the connector object (see section 5.7.4).

```
rebind_connector(<connector_id>,<component_id>);
```

maps to:

```
ArchReflect::rebind_connector(<connector_id>,<component_id>);
```

The second architectural adaptation action allows the replacement of components:

```
replace_component(<component_id>,<component_id>);
```

maps to:

```
ArchReflect::replace_component(<component_id>,<component_id>);
```

The jitter operation in the ACDL is used to prevent adaptation actions from being executed too frequently. In the current implementation it causes the adaptation contract object to sleep for a user defined period of time. The sleep operation is defined in a base class of the adaptation contract, `JTCThread`.

```
jitter(<milliseconds>);
```

maps to:

```
sleep(<milliseconds>);
```

RL and CRL Policies

The RL policy contains a number of declarations that

```
rl_state <rl_state_name>(<state_name> [,predicate]);
```

maps to the struct

```
struct rl_state {
   rl_state(string s, string p) { ...}
   string state_name;
   string predicate;
   boolean is_feedback_state;
};
```

The RL policy contains a number of declarations that are used by RL. Note that actions cannot take a priority as a parameter, as they are invoked directly on components in order to receive a reward, rather than being invoked via the conflict resolver that doesn't supply a reward. There is only one RLAlgorithm supported in the current implementation, a model-based RL algorithm based on dynamic programming (Sutton and Barto, 1998).

```
ListStates = {<rl_state_name1>,..,<rl_state_nameN>};
ListActions = {<actionID1>,...,<actionIDN>};
rl_policy <agent_name>(ListStates,ListActions,RLAlgorithm);
```

maps to:

```
const char* ListStates[] = { "<rl_state_name1>",  ..., "<rl_state_nameN>"};
const char* ListActions[] = { "<actionID>",  ...,"<actionIDN>"};
rl_policy agent_name(ListStates, ListActions);
```

where `rl_policy` is a subclass of a class `MDP` that stores the set of `rl_state` structs, states and actions in the MDP. The `MDP` class also contains a mode of the state transition counts. A RL policy is started execution using

```
start_mdp(<agent_name>, <num_trials>);
```

that maps to a method defined on the `MDP` class:

```
void start_mdp(rl_policy& rl, int num_trials=-1);
```

CRL extends RL with decay:

```
decay <decay_name>(<state_name>, <scaling_factor>, <td>);
```

Decay is implemented as a local feedback event, and a function is inserted in `contract::init` that creates the predicate descriptor for the decay declaration using the `configuration` interface:

```
string <decay_name> = configuration_manager::
   create_decay_descriptor("<decay_name>",<scaling_factor>,<td>);
```

The local feedback events for decay and advertisement are generated from the ACDL code below, where the handler function `decay_state_name` updates `<state_name>` using the decay predicate: :

```
predicate <decay_name>(<decay_file>);
event event_<decay_name>(<state_name>,<decay_name>,low,<decay_state_name>);
```

CRL also extends RL with advertisement. Advertisement is implemented in the feedback event manager. An advertisement is defined as a predicate on an external component feedback state and it is registered with the feedback event manager using the `configuration` interface, also specifying the internal component feedback state. Similar to decay, the predicate on the remote component feedback state is registered as a feedback event by the feedback event manager, and if the feedback event is raised, the updated value for the internal component feedback state is updated with the newly advertised value for the external component feedback state. So,

```
advertisement <ad_name>(<internal_state>,<external_state>,<seconds>);
```

is mapped to

```
string <ad_name> = configuration_manager::
   create_advertisement_descriptor("<ad_name>",<seconds>);
configuration_manager::register_advertisement
    (”<internal_state>”,”<ad_name>”, ”<external_state>”);
```

5.5.2 Proxies and Pluggable Contracts

Client-side adaptation contracts are packaged with proxies to remote components and loaded dynamically as clients bind to components (see figure 5.23). They are unloaded when a client unbinds from a remote component, thus unplugging the adaptation contract. In this way the adaptation logic for a component's connection can be replaced along with the old proxy to the remote component.

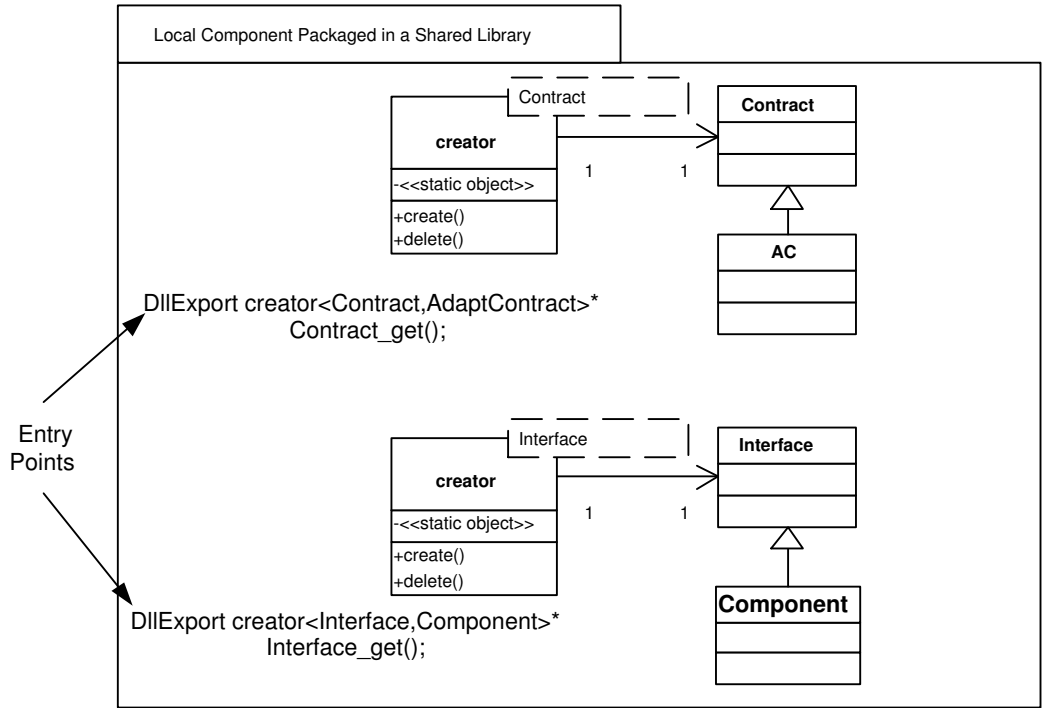


Figure 5.23: KOM packaged component.

5.6 K-Component Framework

The AMM consists of several components - the AMM-DOM, a XML DOM representation of the configuration graph of components and connectors, and the KOM registry of references to component and connector objects. The Registry provides the representational link from the AMM in the configuration graph to the actual component and connector objects. The main interfaces that are used to access and update the AMM are the `ArchEvents` interface in the configuration manager that is used to reify architectural events in the application level and `ArchReflect` MOP that is used by adaptation contracts to monitor and reconfigure the AMM.

5.6.1 Configuration Manager

A configuration manager is a singleton deployed in every K-Component runtime that provides bootstrapping and management services. It provides services such as a thread manager, an adaptation contract manager and a configuration service for initialisation and termination of the meta-level facilities (see figure 5.24). The configuration manager provides the following standard interfaces to components and connectors:

- `Configuration` interface to startup/shutdown both services and application threads
- `ArchEvents` interface to register and deregister components and connectors to/from the AMM

The configuration manager implements a threading model that must be followed by all threads in a K-Component runtime, including application-level components. The common threading model allows a thread manager component to provide services to start, stop and terminate application and meta-level threads. The configuration manager also contains the adaptation contract manager and feedback event service that are discussed in section 3.7.

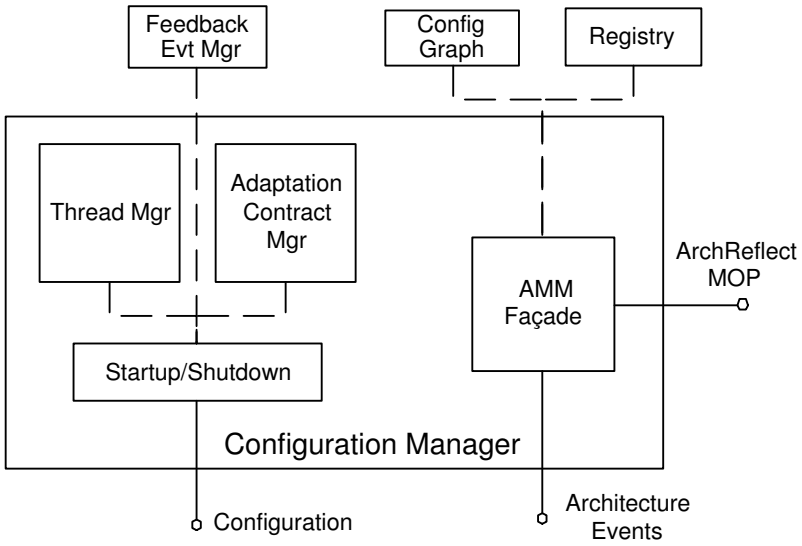


Figure 5.24: The configuration manager and sub-components.

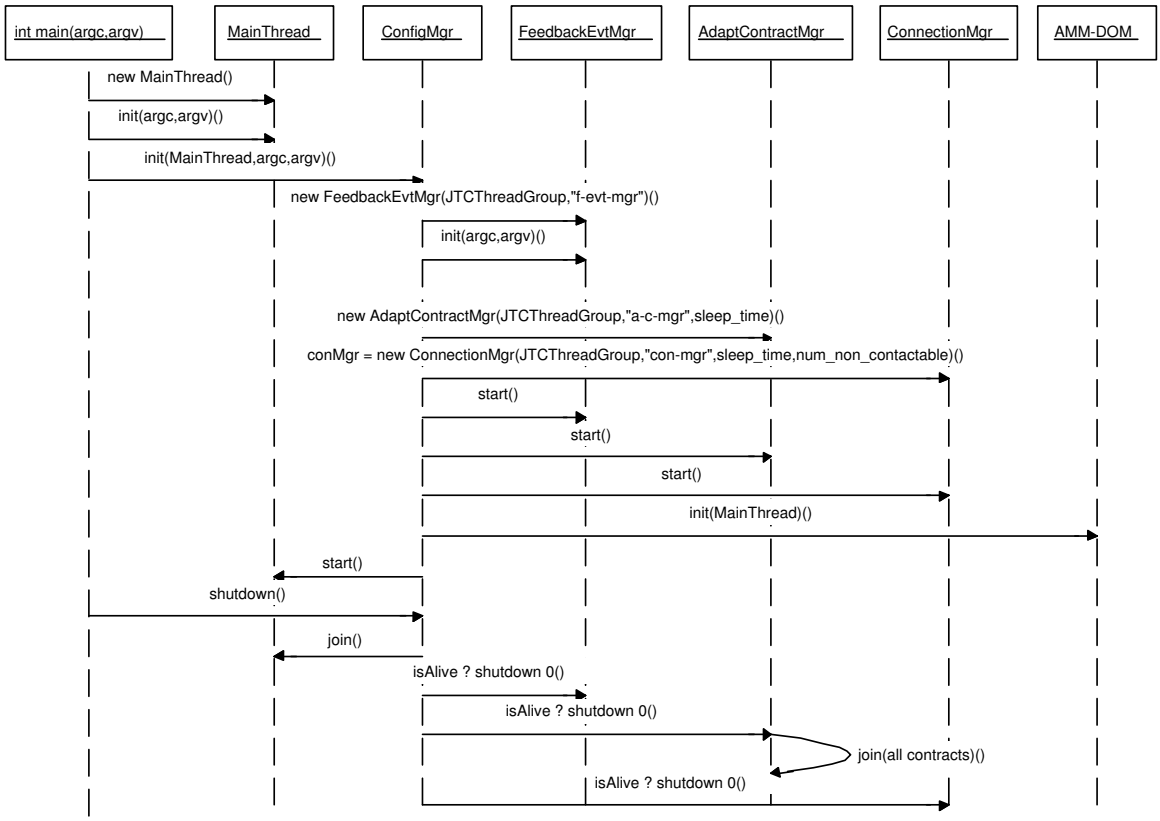


Figure 5.25: Configuration manager startup and shutdown.

The configuration manager also provides a facade interface, called ArchReflect, that uses the AMM and KOM registry services when updating the AMM or searching for component and connector instances.

Configuration Interface

The `configuration` interface is used by application programmers to initialise the runtime, launch application threads, register/deregister adaptation contracts and shutdown the runtime (see figure 5.25). The main service provided by `configuration` interface is thread management. The threading model is provided by the package JThreads 2.0.1 (Concepts, 2000), a framework that provides a Java-like API for writing multi-threaded programs in C++. Applications are started as a JThread object from the mainline by passing the application thread object to `configuration::init()`. The configuration manager and the main thread of control can then wait for the application thread to exit by calling `configuration_manager::shutdown()`, which also stops all other system threads before allowing the runtime to exit. Some of the limitations of the JThreads implementation include the lack of a guarantee on which thread will be woken up after a thread yields the processor, so theoretically there is potential for starvation and the lack of support for thread suspension.

ArchEvents Interface

The `ArchEvents` interface provides operations that allow the synchronous reification of events that occur at the application level. The `ArchEvents` interface is a facade interface implemented by the configuration manager that provides update operations to the AMM that are called by components and connectors. The calls to `ArchEvents` such as `register_component` and `register_connector` are in code generated by the K-IDL compiler that is located in intercession points in components and connectors, e.g., in the connector constructor.

5.7 ArchReflect MOP

The `ArchReflect` MOP represents the interface used by adaptation contracts to asynchronously reflect on the operating state of components and connectors using feedback states. `ArchReflect` is a facade interface implemented by the configuration manager that provides reflective operations to adaptation contracts (see figure 5.24). The monitoring operations in `ArchReflect` only use the configuration graph, while adaptation operations use both the configuration graph and the KOM registry. On AMM update events, such as component/connector creation/deletion or connector binding/unbinding, the `ArchReflect` MOP updates both the configuration graph and the KOM registry, ensuring that consistency between them is maintained. For updates to the AMM with remote components (either incoming or outgoing), only the configuration graph is updated. Updates are performed on component binding by transferring component descriptions (see figure 5.26).

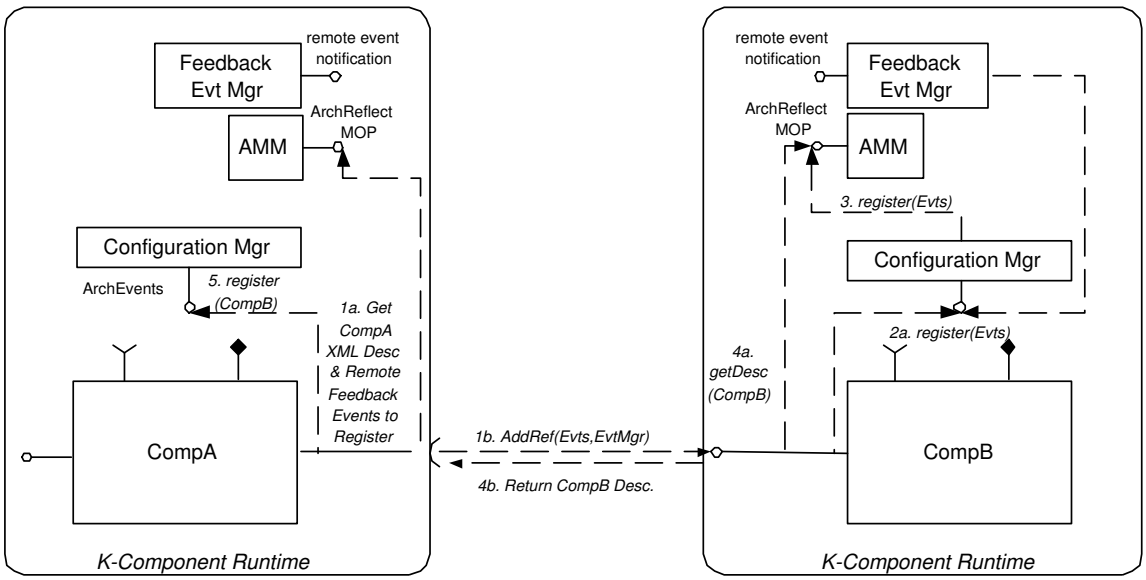


Figure 5.26: Automatic construction of the AMM and registration of remote feedback events.

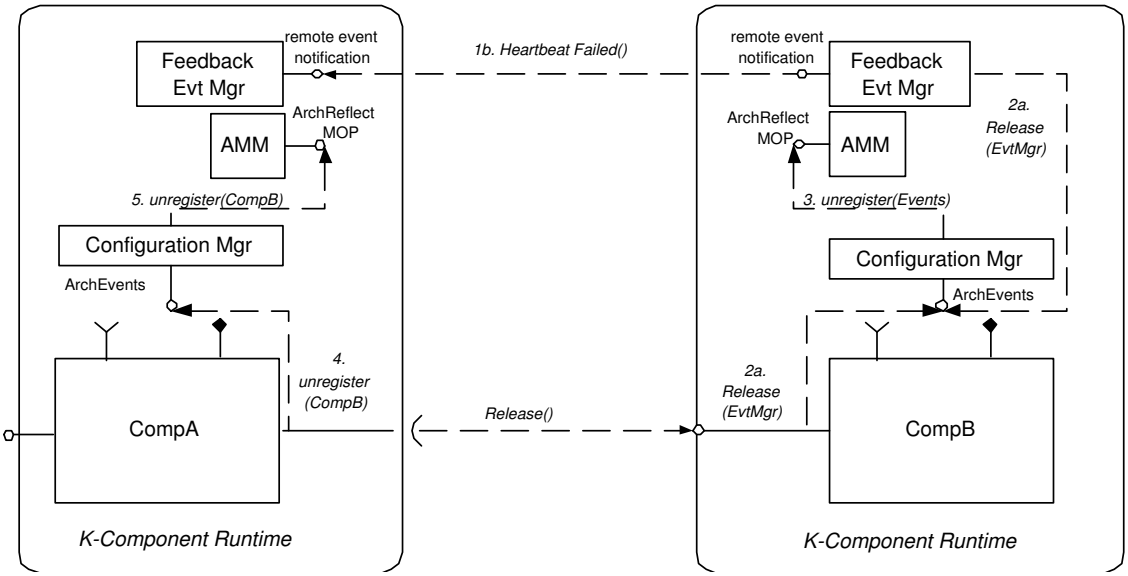


Figure 5.27: Deregister a client's feedback event manager and feedback events.

5.7.1 Automatic Generation of the AMM Configuration Graph

The AMM configuration graph is represented as a XML-DOM object, built using Apache Xerces for C++. A default DOM object that contains a `karchitecture` element is created when the AMM is initialised on startup. The DOM object is updated as components and connectors are created, and connectors bind to components and unbind from components. Component descriptions are generated by the K-IDL compiler (see section 5.2) and on component creation they are loaded from the Repository. When a connector binds to a remote component, the component description can be either loaded from the Repository, if available, or received from the remote component on calling AddRef (see figure 5.26). The DOM object is then updated with the new component description.

5.7.2 KOM Registry

When an update is performed on the AMM, it needs to be reflected in the base-level components and connectors. Since the configuration graph only provides a representation of the components and connectors, a mechanism is required to reflect updates to component and connector objects. The causal connection between the AMM and the components and the connectors in a K-Component runtime is implemented using a Registry of `Object` instances.

The Registry is a store for references to component, connector and creator objects. It provides a lookup interface that can, given an identifier, return a reference to an active component, connector or creator object. Components are stored in the Registry in a `multi_map` data structure.

The Registry stores references to components and connectors and it is updated using the `ArchEvents` interface and can be queried using the `ArchReflect` interface.

The Registry of objects is defined as:

- $O_k = \{o_1, ..., o_N\}$, where $o_i = \{id,\ k_i\}$, where k_i is a reference to an object that may be a component, c_i, or a connector, l_i, and id is a stringified identifier.

The Registry also stores a list of creator objects for KOM objects. As creator objects are responsible for creating and deleting objects from shared libraries, they need to keep track of the objects they've created. The Registry of creator objects is defined as:

- $F_k = \{\{creator_0, C_0, d_0, id\}, ..., \{creator_M, C_M, d_M\}\}$, where $creator_i$ is the shared library's creator object used to create all component objects, $O_i \in C_i$, from the shared library $d_i \in D_k$, where D_k is the set of all explicitly loaded libraries in the runtime's address space.

The Registry provides a lookup service to retrieve references to component, connector or creator objects. The lookup service provides the representational link between the component and connector identifiers in the configuration graph and the KOM object references in the Registry.

5.7.3 Component Replacement

Components can be replaced by adaptation contracts that call the replace component operation defined on the ArchReflect MOP:

```
double ArchReflect::replace_component(const char* CompID, const char* kref);
```

The component replacement operation has to guarantee structural integrity, ensure interacting components are in mutually consistent states and maintain application invariants (see figure 5.28). In K-Components, a reconfiguration protocol has been implemented that meets the structural integrity and mutual consistency requirements:

1. invoke a `block_for_reconfiguration` method on the incoming connector that needs to be in a consistent state before the component can be replaced.

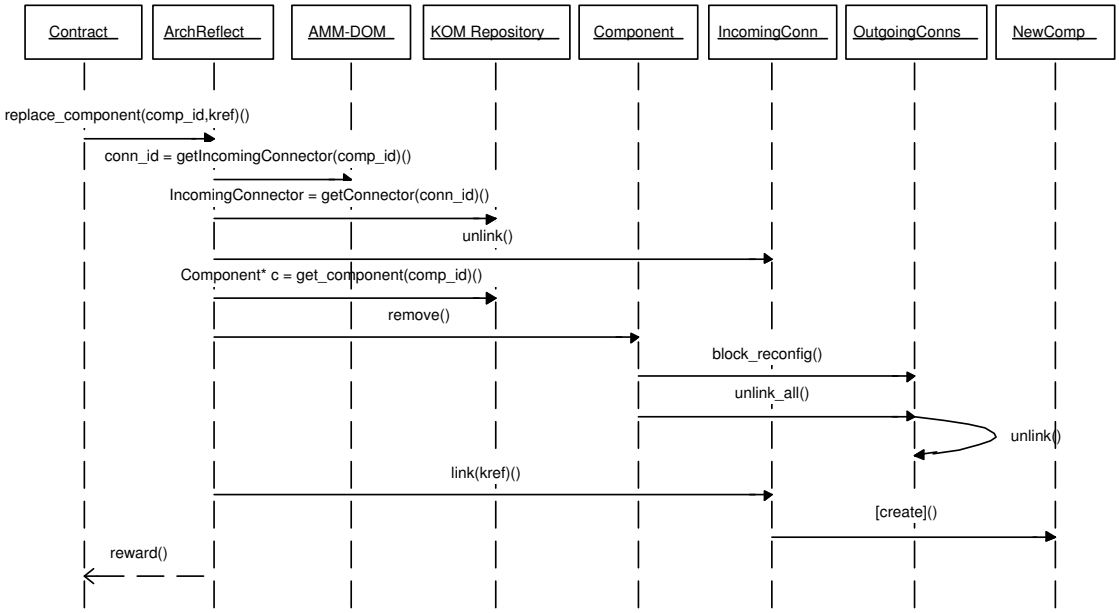


Figure 5.28: Component replacement operation in an adaptation contract.

2. Call `remove` on the component, that forces the incoming connector to wait for ongoing computation and communication in the component's outgoing connectors to finish. The component does this by calling the `unbind` method on all its connectors. The threading model assumes that no new threads of control are allowed to be initiated in the component while it is being replaced. There is an assumption here that both component computation and interactions end in bounded finite time.

3. The component is deleted and a new component is created, possibly as a KOM object. The reference to the old component is replaced with the new component in the incoming connector.

In the K-Component reconfiguration protocol for component replacement, only the component being replaced is forced to a reconfiguration safe state prior to being replaced. Components that initiate communication with a component under reconfiguration are blocked at the incoming connector, but they can take their own decisions as to whether they will wait for the component to be replaced or to rebind to a different component that provides the same service. State transfer between the old component and the new component can be facilitated by infrastructure, but it requires the presence of a copy constructor for the component.

5.7.4 Reconfigurable Connectors and the Reconfiguration Protocol

Connector objects provides binding (see figures 5.29, 5.7) and unbinding (see figures 5.30, 5.8) operations that can be called from adaptation contracts (see figure 5.31).

Connector unbind operations can be invoked by adaptation contracts in a different thread of control from a component handling requests from clients. The reconfiguration of a component's connectors must be thread-safe and not affect system integrity. Ongoing computation and communication in connectors, as well as state information at the component-level and middleware-level should be in a

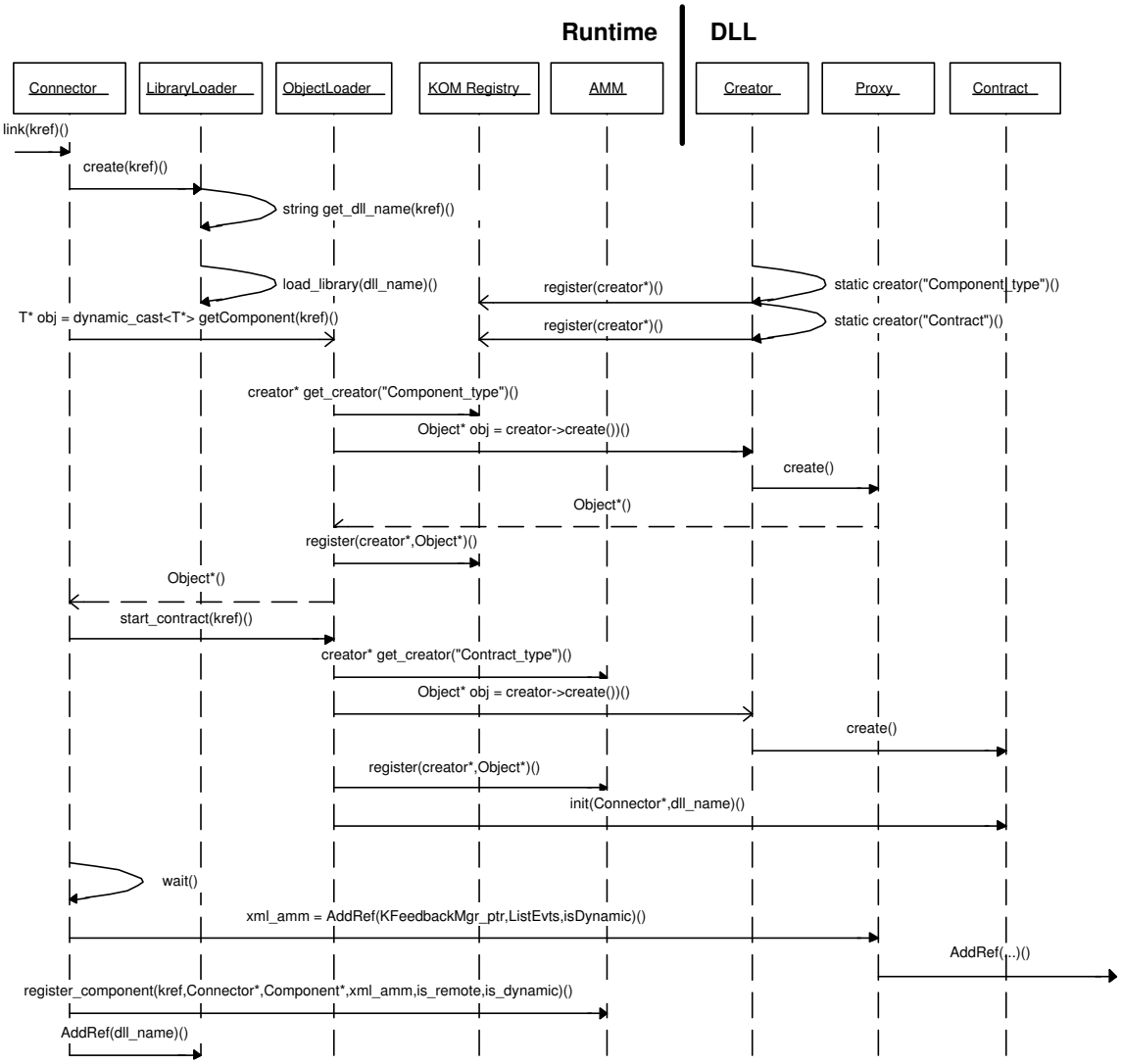


Figure 5.29: Client-side binding.

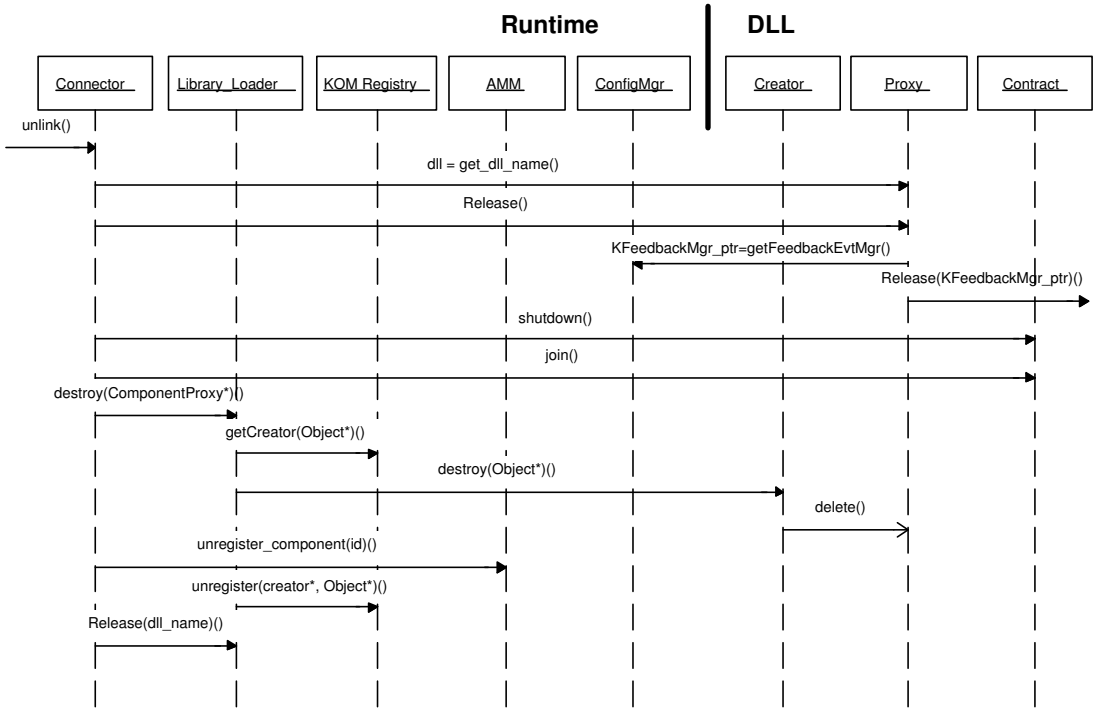


Figure 5.30: Client-side unbinding.

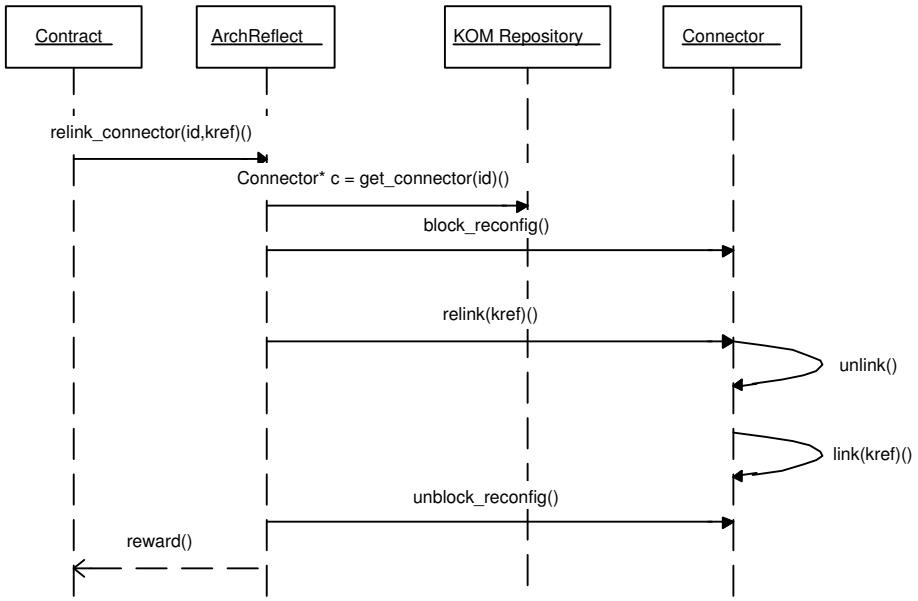


Figure 5.31: Connector reconfiguration operation.

mutually consistent state with a server before connectors are unbound. At the middleware level, the transport protocol for the CORBA proxy is the stateless (OMG, Dec. 2002) that allows connections to be broken without affecting system integrity. At the component-level, however, registered feedback events and references to feedback event managers at servers mean that the client must inform the server of disconnection in order to enable the server to deregister feedback events and the feedback event manager reference on connection unbinding (see figure 5.8). Reconfigurable CORBA bindings that meet the adaptation consistency requirements from section 2.4.3 have been demonstrated in previous projects (Almeida, 2001; Batista et al., 2003; Sadjadi and McKinley, 2004), and a similar approach is followed in K-Components.

In K-Components, a reconfiguration protocol that is based on a RPC-consistency model is supported (Almeida, 2001). The reconfiguration protocol ensures that a connector has reached a safe state before it can be reconfigured. A connector is considered to enter a reconfiguration safe state when it has no ongoing computation or communication. There is no attempt to drive the source or target component to a safe state. An assumption of the reconfiguration protocol is that interactions between components finish in bounded time (Wermelinger, 2000; Moazami-Goudarzi, 1999), enabling the reconfiguration protocol to lock connectors before reconfiguration is performed.

The reconfiguration protocol has to ensure that no new communication is initiated on a connector that is going to be reconfigured. Connector operations `inc_invoke_count` and `dec_invoke_count` are called as pre-operations and post-operations on all public methods defined on the connector. They are used to increment and decrement a reference counter that indicates the presence of ongoing communication in connectors, as well as a means to block a connector from initiating communication. The reconfiguration protocol blocks a component from invoking an operation on a connector that is being reconfigured by locking a synchronisation object using `block_for_reconfiguration` method. This causes components that enter `inc_invoke_count` to wait until reconfiguration has completed, and the reconfiguration protocol has notified the component's thread by invoking the `reconfiguration_complete` method. Synchronisation functionality comes from the JTCThreads package (Concepts, 2000).

Finally, the reconfiguration protocol avoids circular referencing problems associated with other component models, such as COM (Microsoft, 2002), as connectors only wait for ongoing communication to terminate before they can be reconfigured. Components do not maintain reference counts for connected clients. It is the responsibility of adaptation contracts in affected clients to identify broken or dangling connectors to components and to reconfigure the connectors to bind to an appropriate component.

5.7.5 Adaptation Contract Manager

Each K-Component runtime has an adaptation contract manager that is responsible for managing the list of active adaptation contract threads, and synchronising their interaction with the AMM. It is responsible for starting and stopping adaptation contracts.

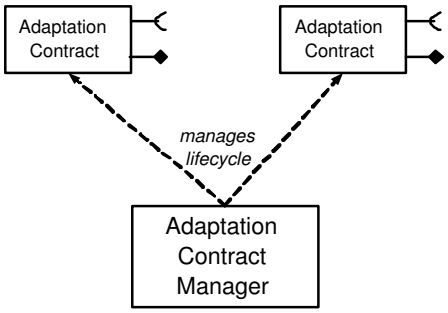


Figure 5.32: Adaptation contract manager.

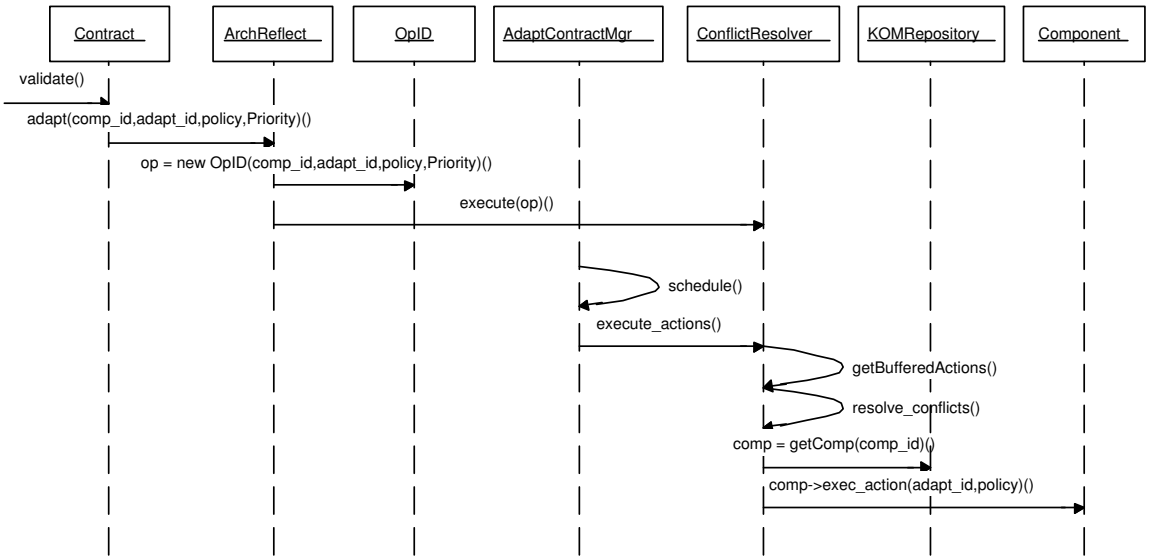


Figure 5.33: Component adaptation action execution with conflict resolution.

Conflict Resolver and Adaptation Action Rewards

The conflict resolver is an autonomous, singleton component that stores a queue of adaptation actions, traverses the queue looking for conflicting actions and then resolves any conflicts and executes the actions. The adaptation actions are sent to the conflict resolver by adaptation contracts using a design based on the Active Object Pattern (Lavender and Schmidt, 1995), which decouples adaptation action requests from adaptation action execution allowing the conflict resolver, which resides in its own thread of control, to decide on which actions to execute. The priority parameter is used by a conflict resolver to resolve conflicting adaptation actions. The default policy of the conflict resolver for resolving conflicts is to execute the action with the highest priority and drop all action requests with lower priorities. If two or more action requests have the same, highest priority, then the first action in the queue is executed.

5.7.6 Feedback Event Manager

The feedback event manager is a singleton component deployed in every K-Component runtime that acts as both a producer and consumer of feedback events. A *SyncMgr* thread in the feedback event manager is responsible for identifying raised events and notifying them to the appropriate feedback

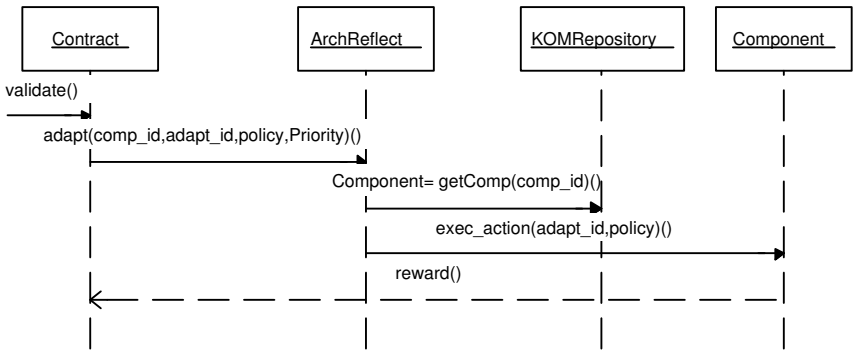


Figure 5.34: Component adaptation action execution with reward model.

event manager. Feedback events can be notified at any time and there is no assumption of any ordering on events or synchronisation of managers using global clocks.

Feedback events are defined in adaptation contracts and are registered with the AMM. Feedback events that refer to remote components, remote feedback events, require a two-part registration process. Firstly, the feedback event is registered with the AMM in the local runtime when its adaptation contract is loaded and initialised. The registered event consists of the target component feedback state, a predicate descriptor, event priority and a handler to execute when the event is raised. Secondly, the feedback event is registered with the AMM in the remote runtime. This is done by adding a feedback event registration phase to the binding of the outgoing connector to the remote component. Feedback events are associated with outgoing connectors, in order to be identified for the registration phase, by specifying the connector as a parameter to the adaptation contract definition (see section 3.7.3).

The feedback event manager also provides a SyncMgr thread that performs several tasks. These include the synchronisation of feedback states between components and connectors and the AMM, the matching of predicates defined on events and the notification of feedback events. When a predicate for a remote feedback event is matched, the SyncMgr for the remote runtime notifies the local feedback event manager of the updated value of the component feedback state, which the local feedback event manager uses to update the local AMM. When the SyncMgr in the local runtime is next scheduled it raises the local feedback event, registered in the first part of the registration process. Its associated handler for the event is then executed. Feedback events can be given a priority, to enable the resolution of conflicting adaptation actions requested by different handlers. The default conflict resolution strategy is to not execute the lower priority conflicting actions.

Feedback notification comes in the form of CORBA data objects that do not contain references to components or connectors. The feedback event consumer part of the feedback event manager is implemented as a CORBA callback mechanism. Distributed callbacks were favoured over the CORBA Event service due to their ability to provide a lightweight service suitable for decentralised environments. However, distributed callbacks suffer from a number of problems such as callback persistence and tight coupling of clients and servers (Henning and Vinoski, 1999).

Clients register a reference to their feedback event manager with the incoming connector at the

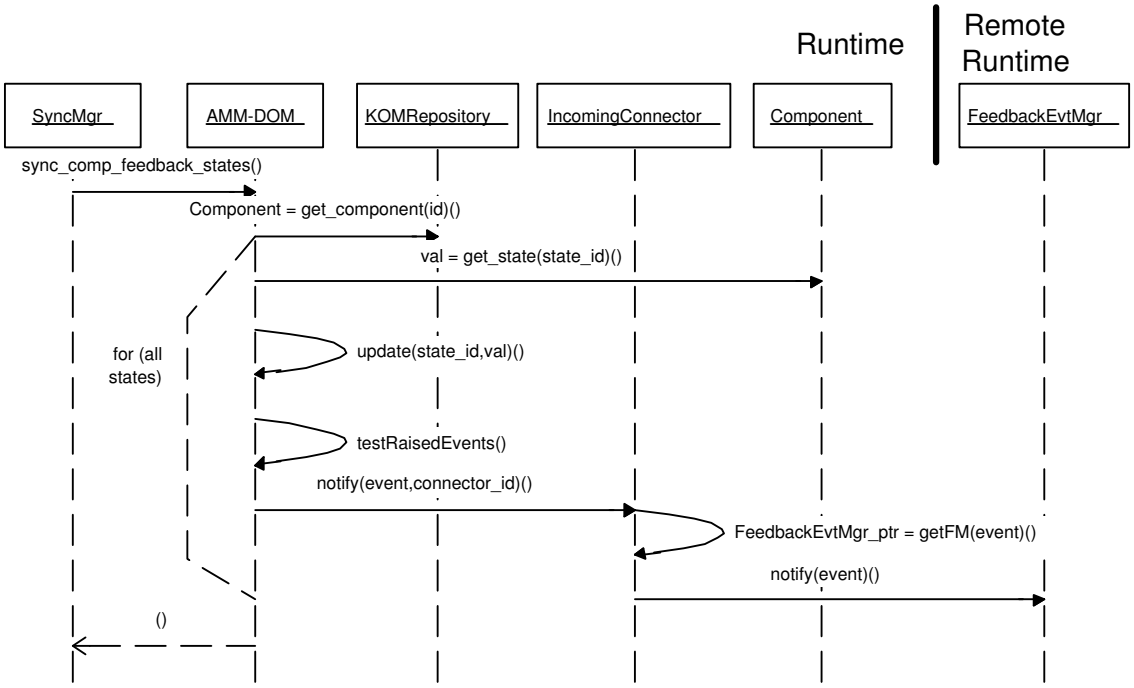


Figure 5.35: Component feedback state synchronisation between components and AMM. Event evaluation and notification.

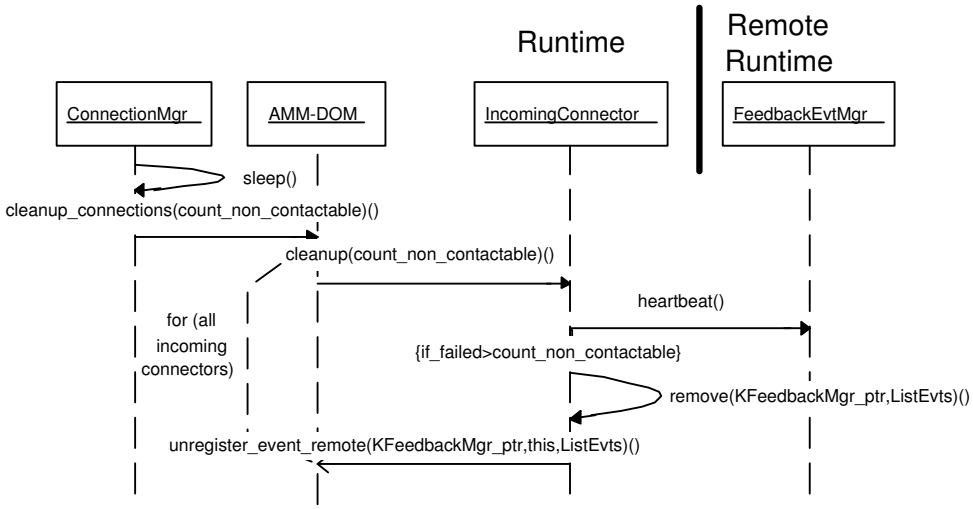


Figure 5.36: Connection manager cleanup thread.

server when a client binds to a server and calls `AddRef(FeedbackEvtMgr_ptr p, ..)`. Orbacus/C++ provides an implementation of `is_equivalent(CORBA::Object)` that can be used to compare object references and test for duplicates[3]. Clients are notified of feedback events using the one-way operation:

```
oneway void k_meta_event_notification(in KEvent k_event)
```

One-way operations are suitable for decentralised environments as they have fire-and-forget semantics. Feedback event managers are not concerned whether the events are delivered or not, as it is the responsibility of every client to reason about its own operation, not the responsibility of a server to manage the client's view of the network.

In a decentralised environment, the availability of clients and servers is subject to frequent change, invalidating references to feedback event managers. A clean-up thread, the Connection Manager (see figure 5.36), is provided that uses the `AMM-DOM` to periodically call `heartbeat` operations on feedback event manager references. If a user-defined count of sequential heartbeats fails, the Connection Manager indicates that a client has dropped its connection and the connection is cleaned up.

Another potential problem that one might think could occur with callbacks is if the client of a feedback event executes handler code that in turn calls the server component and forms some dependency on its inconsistent state (Szyperski, 1998). This problem does not arise, however, as servers execute callbacks as one-way operations, i.e. the server doesn't block waiting for the callback to return allowing dependencies to arise on inconsistent state. Another restriction that prevents this problem is that handler code for feedback events is restricted to accessing and updating the AMM using adaptation contract threads. These mechanisms prevents the occurrence of the cyclic dependency and self-recursion problems associated with callbacks.

[3]The CORBA specification does not require all ORBs to support `is_equivalent` operation under all circumstances, e.g., when objects are indirected through a proxy (Henning and Vinoski, 1999).

```
enum EventPriority { Low, Normal, High, System};
struct KEventRegistration {
        string name;
        string interf;
        string cb_conditions;   // XML Descriptor
        EventPriority priority;
};
typedef sequence<KEventRegistration> KEventRegistrationList;
struct KEvent {
        string name;
        double value;
        EventPriority priority;
};
typedef sequence<KEvent> KEvents;
struct ConnectionStatus {
...
};
interface KFeedbackMgr {
        oneway void k_meta_event_notification(in KEvent k_event);
        ConnectionStatus k_meta_heartbeat(in string connector_name);
        void deactivate();
...
};
```

5.8 Asynchronous Reflection

One of the challenges when implementing the asynchronous reflection model is the requirement for synchronisation between adaptation contracts and application threads when accessing the shared AMM. Many existing reflective programming languages do not address the problem of synchronised access to the shared meta-level as reflective code is executed in a single-threaded environment synchronously with program execution (Schaefer, 2001). There is contention in K-Components for the shared AMM structure (see figure 5.37), although this is reduced by specifying methods as either const (read-only) or modifier methods in C++.

In order to prevent possible thread starvation, adaptation contracts yield the processor at entry and exit points. Yield points can be specified in components by programmers to reduce the amount of time until adaptation contracts are scheduled. For component programming, this allows programmers to specify at specific points where reflective code should be scheduled to execute. For example, when an exception is thrown in the base-level, the programmer may yield the processor to increase the responsiveness of adaptation contracts in reasoning about the system and fixing problem that caused

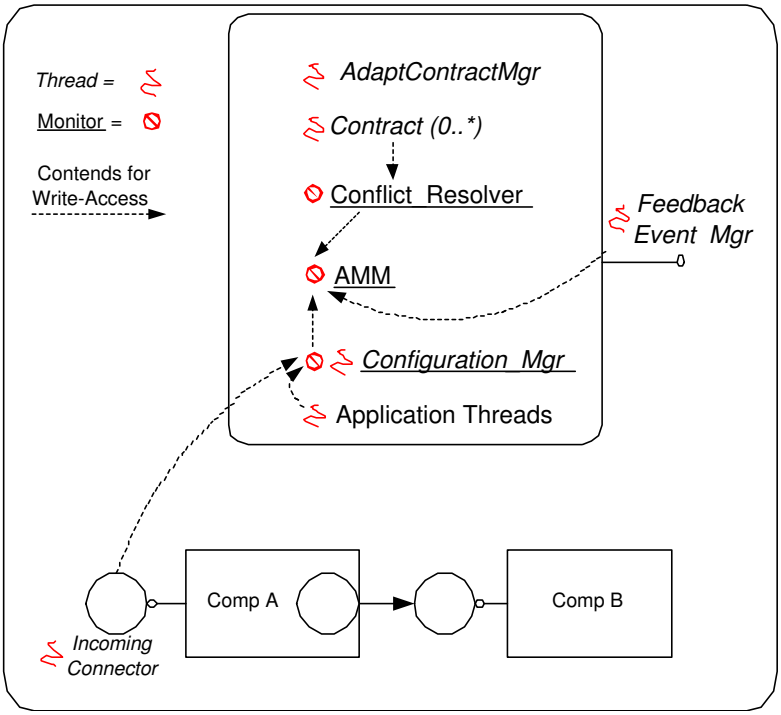


Figure 5.37: Contention for the AMM in the implementation of asynchronous architectural reflection.

the exception.

Reification Categories in the AMM

Reification Category	Reflective Operations	Reification	Meta-Base Consensus
Configuration Graph	Introspect, Reconfigure	Intercession	Strong
Component	Introspect, Action, Replace	Intercession	Strong
Connector	Introspect, Reconfigure	Intercession	Strong
Component Feedback State	Introspect	Feedback Event Manager	Periodic Sync

Table 5.5: Reification categories and the causal connection.

K-Components provides the AMM as a meta-level software architecture that reifies several aspects of the system, known as reification categories. Architectural reification categories represent architectural features that can be introspected and/or modified by the programmer. The configuration graph and the individual components and connectors can be both introspected and modified at runtime, while component feedback states can only be introspected at runtime, (see table 5.5).

Adaptation Agility

The sampling time interval for observations, t_c, is configurable by changing the sleep time for the SynMgr thread (see figure 5.35) on calling `configuration_manager::init(..)`. The scheduling time interval for adaptation contracts, also configurable by setting a sleep time for each contract, affects the adaptation agility (Li, 2000) of the application. There is a trade-off between configuring

the responsiveness of the adaptation contracts in reacting to adaptation conditions and minimising the execution overhead of reflective code. Asynchronous reflection can allow programmers to configure this trade-off. The scheduling time interval for adaptation contracts affects the adaptation agility (Li, 2000) of the application. There is a trade-off between configuring the responsiveness of the adaptation contracts in reacting to adaptation conditions and minimising the execution overhead of reflective code.

5.9 Summary

This chapter described the implementation of a prototype of the K-Component model, with emphasis on the programming model used to write components and their adaptation logic. The model builds on the Orbacus/C++ ORB and compiler to allow programmers to define self-adaptive components in K-IDL and implement them in C++. The ACDL compiler is implemented using ANTLR and the adaptation contract classes output are included in either proxies or runtime projects, where adaptation contract objects are initialised by connector objects.

The programming model hides much of the complexity involved in specifying adaptation logic for components through the support of the ACDL. There are however, some programming restrictions in the model. In particular, applications must be specified as JThread objects and threads must be registered with the configuration manager to enable clean shutdown of the runtime.

The AMM is implemented using both the AMM-DOM, that stores the configuration graph of components and connectors, and the KOM registry, that stores references to actual connector and component objects. The KOM model allows objects to be loaded at runtime from libraries allowing the construction of systems that can evolve at runtime without the requirement for restarting or recompiling the system. It also enables adaptation contracts to be replugged at runtime.

In the next chapter, the K-Component model is evaluated by building a decentralised FileStorage application that can adapt and optimise itself to a changing environment. A self-adaptive FileStorage component is also evaluated that demonstrates self-healing behaviour.

Chapter 6

Evaluation

> "Annual income 20 pounds, annual expenditure nineteen nineteen six,
> result happiness.
> Annual income 20 pounds, annual expenditure twenty pounds, ought and six,
> result misery."
>
> Mr Micawber in David Copperfield by Charles Dickens, (1850)

This chapter reports on the evaluation of CRL, the K-Component model, and the implementation of the K-Component model against the objectives and requirements for autonomic distributed computing systems introduced in Chapters 1 and 2. The objective in evaluating CRL is to show that it can be applied to coordinate the self-adaptive behaviour of K-Components in order to establish and maintain autonomic properties in a decentralised system. At the component-level, the performance of the self-adaptation of K-Components in the face of changes in their environment is evaluated. The evaluation also demonstrates that K-Components provide a structured and modular approach to specifying autonomic behaviour of components by means of the ACDL. The K-Component programming model is also evaluated through analysis of the expressive power, features and restrictions of the ACDL as a language for specifying autonomic behaviour as decision policies. The final part of the evaluation compares the features provided by K-Components with existing architectures supporting self-adaptive software introduced in Chapter 2.

6.1 Evaluation Objectives

A number of requirements for distributed autonomic computing systems were introduced in Chapters 1 and 2. In this chapter, the K-Component model and CRL are evaluated by comparison with how well they fulfil these objectives. The goals of the evaluation come under two main areas:

1. The establishment of autonomic system properties in a decentralised system using CRL and K-Components.

2. K-Components and the ACDL as a model for specifying and implementing self-adaptive components that can exhibit autonomic properties.

The first evaluation criteria is addressed by an implementation and evaluation of a decentralised, file storage system in section 6.2 using K-Components and CRL. A set of experiments are described in section 6.3, assessing how system-wide autonomic properties are realised using CRL and K-Components. The goal in evaluating CRL is to demonstrate the autonomic properties of systems built using CRL and the general applicability of the technique for building autonomic systems, rather than an exhaustive evaluation of how tuning CRL parameters affects a system's operation.

The second evaluation goal is addressed by analysis of the K-Component model as a model for building autonomic components. Firstly, a performance evaluation of K-Components relative to CORBA is presented in section 6.4, then an evaluation of the ACDL as a declarative language for specifying autonomic behaviour using different decision policies is undertaken in section 6.6 and finally a feature-based comparison of K-Components with existing models for building self-adaptive and autonomic systems is described in section 6.7.

6.2 Decentralised Load Balancing

In this section K-Components and collaborative reinforcement learning are applied to the area of adaptive load balancing (Schaerf et al., 1995). Load balancing is the problem of efficiently using the resources in a distributed system to process load(s) generated by some node(s). It is an example of a distributed decision problem, and previous solutions for decentralised environments have been based on numerical performance measures and decentralised optimisation. Existing load balancing systems for decentralised environments (Schaerf et al., 1995; Jelasity et al., 2003; Montresor et al., 2002; Rao, 2003; Othman et al., 2001) have been mostly peer-to-peer systems that contain important differences from the system described in this section. Firstly, they are not built using a component model, as they are usually implemented as (overlay) routing protocols. Secondly they exploit the bidirectionality of network connections to transfer load between neighbouring peers, whereas connectors in component models are typically directed. Thirdly and finally, they are often not adaptive (Jelasity et al., 2003; Rao, 2003), in that they make design assumptions about the environment, such as the existence of a random topology (Jelasity et al., 2003). Metrics used to evaluate these schemes are often based on system properties such as the amount of load transferred between nodes (Rao, 2003), optimal use of system resources (Schaerf et al., 1995) and the equalization of load distributions (Othman et al., 2001; Jelasity et al., 2003).

The convergence between load levels at different components in a load balancing system (or load equalisation) is a widely-used valid metric for the evaluation of a load balancing system (Montresor et al., 2002), although in dynamic, uncertain environments where the availability of components and network connections is in constant flux, the ability of agents to adapt their policies to discover better solutions as the environment changes is an important goal for the load balancing system (Dowling et al., 2005). Experiments 1 to 6 in this chapter address CRL's ability to perform adaptive optimisation in the decentralised load balancing system. The other evaluation criteria used for the decentralised file storage application is the establishment of system-wide autonomic properties using only local knowledge at the component level. Desirable autonomic properties of the system include the optimisation of an unpredictable set of system resources through adaptive load balancing, and the reconfiguration

of the system to recover from errors or to exploit newly discovered resources.

6.2.1 Properties of Decentralised Load Balancing

Load balancing is an interesting evaluation problem for an autonomic computing system as the automated load balancing of resources helps a distributed system to optimise its performance. In a decentralised autonomic system, resource consumption and balancing decisions are taken at the component level and autonomic properties should be established at the system level. The load balancing system in this chapter addresses two different properties of decentralised systems:

- Uncertainty: each component does not have up-to-date information about the current load levels at other components in the network, so load balancing decisions are taken locally with uncertainty about the outcome of the action.
- Dynamism: Components join and leave the system and the storage capabilities of components change frequently. Load balancing agents should be able to adapt their decision policies to a dynamic environment.

6.2.2 Design of the Decentralised File Storage System

The load balancing application presented in this chapter has been implemented as a prototype and is not designed to provide features required by real-world file storage systems, but rather to demonstrate the ability of a decentralised system, based on K-Components and CRL, to load-balance files without recourse to centralised coordination or global knowledge. The implemented system makes several assumptions to simplify the problem domain:

- Each component is considered a manager of a set of file storage resources[1]. In the system, every K-Component provides a fixed amount of secondary storage space that can be used to store files. Each K-Component has an in-memory buffer of unlimited size that is used to store files until either the file is stored locally or it is forwarded to a connected component.
- Load (or files in the system) is generated by clients external to the system and each load is of fixed size[2]. Two different models for generating load that are investigated are a peak model and a more linear model.
- Adaptation contracts, based on CRL, decide whether files in a component's buffer should be stored locally or forwarded to a connected component. Once a file is stored locally, it can only be removed by the client that requests the file. There is also no support for replication of files in the system and if a component fails, the files stored at that component and forward references to other files are lost.

[1] Kephart's model of an autonomic management component includes a role as a manager of a set of resources (Kephart and Chess, 2003).

[2] This assumption simplifies the extraction of numerical performance measures from the system. In real-world instrumented distributed systems the extraction of numerical performance measures is an open problem being addressed by the autonomic computing research community, e.g., IBM's Autonomic Toolkit (IBM, 2004).

- K-Components operate in a trusted environment where they are expected to supply honest rewards and adaptation contracts, and are not expected to follow greedy policies that may be detrimental to the interests of the system as a whole.
- The information available to an adaptation contract when selecting a load balancing operation is based on prior experience and advertisements received from its connected components. The contract does not make use of design-time knowledge, e.g., about the capacity levels of its local resources or the arrival rate/size of files received, in order to improve load balancing performance.

6.2.3 File Storage K-Component and Load Balancing Adaptation Contract

The file storage system is composed of `FileStorage` components (see table 6.1) that provide an IDL interface allowing clients to submit and retrieve files from the system. The implementation of the FileStorage component makes use of private methods to store files to the component's local storage area and forward files to a connected component.

The component's interface includes three connectors to other components in the system in order to enable files to be forwarded to remote components. Three was chosen as the outgoing degree for the `FileStorage` component in order to increase the number of network paths in the system and to improve the ability of components to discover less lightly loaded components in the system.

Four states and fours actions are defined on the `FileStorage` component interface and they are used by the adaptation contract to load balance files that are in the component's buffer. The component's `load` state provides feedback on the current load level for the component's file storage, while the `buffered` state provides feedback on the current number of files in the component's buffer. The component programmer updates the values for these states upon receiving files, forwarding files, and storing files to persistent storage. The actions `store` and `forward0..2` are also defined on the component interface. There are three `forward` actions defined on components, each of which load balances files to one of the three components connected to each component. Both `store` and `forward` actions can be executed by the adaptation contract to attempt to store or forward a file in the component's buffer.

The states and actions defined on the component are used by the adaptation contract in table 6.2 to specify the problem of determining the optimal storage location for files in the component's buffer as a CRL policy. Each file that is received by a component results in the value of its buffered state being incremented and a MDP for the CRL problem is started by the `if-then` rule being matched (see table 6.2). A decay model for cached Q-values for the `forwarded` state is also specified, but no advertisement function is specified as it is implemented as a synchronous RPC function in the load balancing system. This can be seen in the pseudo-code for the submit method of the `FileStorage` component in figure 6.1.

```
// IDL
typedef sequence<octet> BinaryFile;
interface File {
   double submit(in string name, in BinaryFile contents);
   Binaryfile retrieve(in string name);
};
// K-IDL
component FileStorage {
  provides File;
  uses File n0;   // 3 connectors to other File Storage Components.
  uses File n1;
  uses File n2;
  state load;     // Current load level of file storage at the component.
  state buffered; // File in Buffer.
  state stored;   // File Stored.
  state forwarded;// File Forwarded.
  action forward0;
  action forward1;
  action forward2;
  action store;
};
```

Table 6.1: K-IDL Definition for the FileStorage component.

```
incoming LoadBalance(){
  ListStates crl_states = {FileStorage::buffered,
          FileStorage::forwarded, File Storage::stored};
  ListActions crl_actions = {FileStorage::store, FileStorage::forward0,
          FileStorage::forward1, FileStorage::forward2};
  decay fs_decay(forwarded, 1.01);
  crl_policy lb(crl_states,crl_actions, fs_decay); //RPC Advertisement
  if (ArchReflect::poll_event(FileStorage::buffered) > 0)
      start_mdp(lb); // until MDP termination
};
```

Table 6.2: ACDL definition of the CRL load balancing policy.

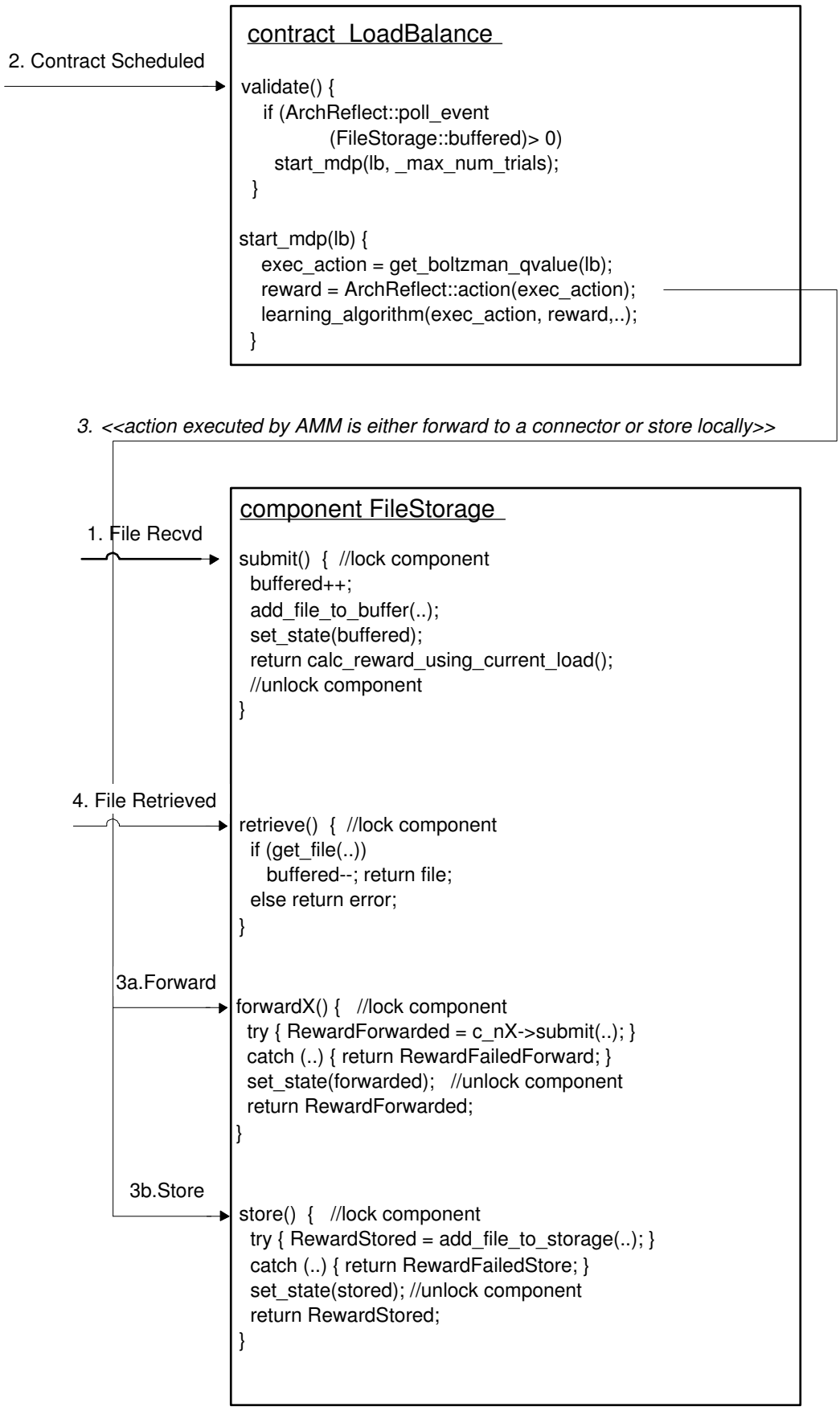


Figure 6.1: Pseudo-Code of LoadBalance Contract and FileStorage Component

6.2.4 Overview of Load Balancing using CRL

Figure 6.2 provides an illustrative guide to the normal operation of the file storage system. Firstly, a file is received by component A and placed in its buffer. The CRL decision policy in the adaptation contract for component A computes that a connected component C that has a lower total *advertised load cost* (the advertised cost plus the connection cost) than A's local storage cost and, with high probability, executes a delegation action defined on Component A that forwards the file to component C. An adaptation contract for component C subsequently stores the file locally at component C. In effect, each component solves a discrete optimisation problem to decide whether to attempt to store or forward the file.

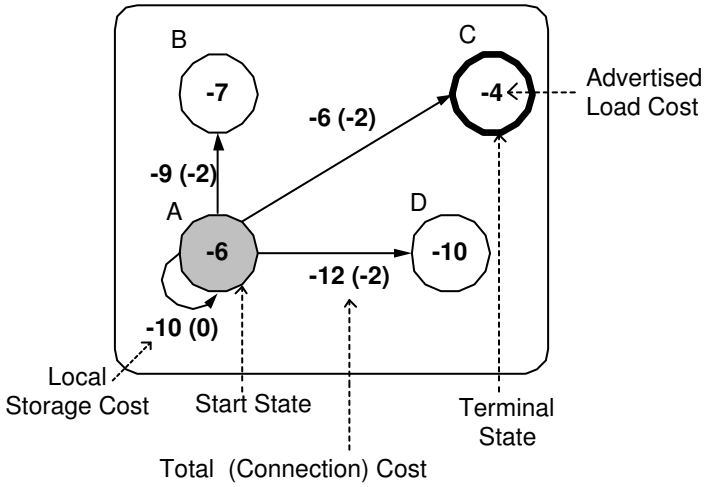


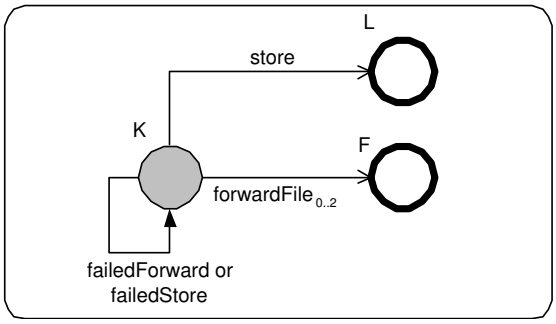
Figure 6.2: Decentralised load balancing decisions in CRL.

6.2.5 Definition of the Load Balancing Application as a CRL System

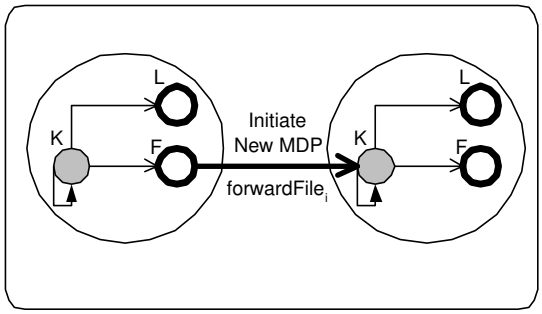
This section defines the load balancing application more formally as a CRL system. The load balancing problem is described as a Markov Decision Process with three states (see figure 6.3): K, L and F. In the K-IDL definition of the FileStorage Component, states K, L and F correspond to the states `buffered`, `stored` and `forwarded`, respectively. State K is the start state for the MDP and indicates that there is a file in the component's buffer that should be either stored locally or forwarded to a connected component. States L and F are terminal states of the MDP. The goal of the adaptation contract, a CRL agent, is to select the `store`/`forward` action that optimises the system problem of load balancing the file in the system. The adaptation contract terminates the MDP by successfully executing a `store` or `forward` action, and making a transition to the terminal state `stored` or `forwarded`, respectively. There is only a single `forwarded` state that is reached if any of the forward actions (`forward0..2`) is successfully executed.

After the execution of an action by the adaptation contract, the distributed reinforcement learning algorithm from CRL is used to update the adaptation contract's policy. The file storage application is formally defined as a CRL system below:

- An adaptation contract n_i has a MDP that consists of a set of states $\mathcal{S}_i = \{K, L, F\}$ corresponding to states defined on the file storage component c_i, where K indicates that a file is in



(a) MDP at an Adaptation Contract



(b) Connected States

Figure 6.3: The FileStorage MDP.

a buffer waiting to be handled, F indicates that a file has been successfully sent to a connected component for storage and L indicates that the file has been stored locally at component c_i.

- The external and internal states are: $Ext(n_i) = \{K\}$ and $Int(n_i) = \{L, F\}$. $\{K\}$ is the start state of the MDP and the terminal states are $\{L, F\}$.

- The connected states are $\mathcal{C}_{n_i n_j} = \{K, F\}$, where a state transition to state F at contract n_i initiates a new MDP at state K for contract n_j (see figure 6.3).

- The actions available at n_i are $\mathcal{A}_i = \{forward_0, forward_1, forward_2\} \cup \{store\}$. The set of delegation actions is $\mathcal{A}_{d_i} = \{forward_0, forward_1, forward_3\}$ where the component, c_i, associated with n_i has three connected (or neighbour) components, c_0, c_1 and c_2, and the action $forward_j(c_j)$ represents an attempted `submit` operation on component c_j. A DOP action, $\mathcal{A}_{p_i} = \{store\}$, is also available that represents the `store` action defined on component c_i. An additional *discover* action is provided to allow the discovery of components, but it is implemented as a static naming service and is, therefore, not included in the MDP or learning strategy.

- Advertisement updates are implemented as synchronous responses to `forward` actions that return V values to clients. An adaptation contract n_i receives V_j advertisements over a connector l_{ij}, updates r_j in its $Cache_i$ for $Q_i(s, a_j)$. It also recalculates the $Q_i(s, a_j)$ value using the advertisement received.

- The distributed RL algorithm used in CRL is based on model-based reinforcement learning and requires a state transition model. The state transition model is implemented as a statistical model of the estimated probability of a *store* or $forward_{0..2}$ action succeeding, similar to the

model in (Curran and Dowling, 2004). In order to build the statistical model, the success (or otherwise) of *store* and $forward_{0..2}$ actions executed by each adaptation contract are sampled. The sampled success/failure events are used to estimate the probability of a future attempted $store/forward_{0..2}$ action being successful. The estimated probability of an action succeeding, $Est\left(P(s'|K,a)\right)$, given the current state K is calculated using:

$$Est\left(P(s'|K,a)\right) = \frac{c_S}{c_A} \tag{6.1}$$

where c_A is the number of attempted $store/forward_{0..2}$ actions and c_S is the number of successful $store/forward_{0..2}$ actions that have been executed. A state transition model is maintained for every possible action in state K as:

- the probability of a $forward_i$ action succeeding: $p_{ijS} = P_i(F|K,a_j)$, $a_j \in \{forward_0, forward_1, forward_2\}$
- the probability of a $forward_i$ action not succeeding: $p_{ijF} = P_i(K|K,a_j) = 1 - P_i(F|K,a_j)$, $a_j \in \{forward_0, forward_1, forward_2\}$
- the probability of a *store* action succeeding: $p_{store} = P_i(L|K,store)$
- the probability of a *store* action not succeeding: $p_{full} = 1 - P_i(L|K,store)$
- for all other state transitions and all actions, $P_i(s'|s,a) = 0$

- For the purposes of the experiments in this thesis, it is assumed that the cost of a connection for all connectors in the system is equal, although in many distributed systems this would not generally be the case. As a consequence, the connection costs for the load balancing experiment are set to a fixed value for all connectors. The following are the connection cost models for the delegation actions:
  $D_i(F|K,a) = r_S \in \mathbb{R},\ where\ a \in \{forward_0, forward_1, forward_2\}$
  $D_i(K|K,a) = r_F \in \mathbb{R},\ where\ a \in \{forward_0, forward_1, forward_2\}$
  $D_i(s'|s,a) = 0,\ where\ a \notin \{forward_0, forward_1, forward_2\}$
  The values for r_S and r_F are set to different static values for the experiments in section 6.3.

- The MDP termination costs describe a reward model for terminating a MDP after executing a *store* or $forward_{0..2}$ action. There are two MDP termination cost models for the single non-terminal state, K, in the MDP. The first reward model, $R(K,store)$, for the *store* action is implemented as a *load function* that returns a reward based on the value of the component's `load` feedback state, i.e., the current load at the component. The higher the load at the component, the lower the reward received for terminating the MDP using a *store* action. For the $forward_{0..2}$ actions the MDP termination cost is a fixed value r_C to reflect the assumed fixed cost associated with forwarding the file. The
  $R(K,store) = Load(c_i,\ f_{load}) \rightarrow \{Low, Med, High, Peak\} \in \mathbb{R}$
  $R(K,a_j) = r_C, \quad where\ a \in \{forward_0, forward_1, forward_2\}$

The load function in the application is designed to linearly scale the reward for four different possible levels of load at a component: { Low, Med, High,Peak} . Scaling the reward produced for store actions to the different levels of load at the component can be used to encourage adaptation contracts to exploit storage space when there is low load at the component and also to encourage the forwarding of files to other components when the load level is high. The range of these four levels and the rewards received for store actions are configured by the CRL system designer.

- The decay model, $Decay(r_j) = r_j.\rho^{td}$, is designed to degrade cached values for the load state of connected components. Depending on the expected level of network dynamism, different values for ρ could produce better load balancing behaviour and be obtained through tuning and experimentation.

- Given state K at component c_i and actions $a \in \mathcal{A}_i$, the Q_i values for the *store* and $forward_{0..2}$ actions are calculated using distributed model-based reinforcement learning:

$$V_i(K) = \max_a Q_i(K, a)$$

Substituting the $forward$ and *store* actions into the above formula, the Q-values for the different actions are:

$$Q_i(K, store) = Load(c_i) + p_{store}\left(V_i(L)\right) + p_{full}\left(V_i(K)\right)$$

$$Q_i(K, a_j) = r_C + p_{ijS}\left(Decay(V_i(F)) + r_S\right) + p_{ijF}\left(V_i(K) + r_F\right), a_j \in \{forward_0, forward_1, forward_2\}$$

- When the Q-values have been calculated, the action that is executed by the adaptation contract is selected using the softmax selection strategy, Boltzmann action selection (Sutton and Barto, 1998). This allows suboptimal actions to be selected, albeit with a configurable probability by modifying the temperature T, and thus allowing exploration of suboptimal actions in order to discover less loaded components.

$$P(K, a) = \frac{e^{-(Q_i(K,a))/T}}{\sum_a e^{-(Q_i(K,a))/T}}, \quad a \in \mathcal{A}_i \tag{6.2}$$

6.3 Experiments

The experiments described in this section are based on a set of adaptive components that attempt to globally optimise resource usage in a decentralised file system without the use of global knowledge, and loosely follow the goals of those defined in (Schaerf et al., 1995). The experiments differ, however, from those in (Schaerf et al., 1995), as they address the issue of demonstrating how the system adapts and optimises itself to exploit resources that are dynamically added to and removed from the system. All of the experiments, except one, are based on a peak load model (Montresor et al., 2002), where a single load generator supplies load to the system. The peak load generator model has the characteristics of streaming traffic, such as multimedia traffic (Fitzpatrick, 2001), where constant bit-rate traffic stems from a fixed small number of sources and the experiments investigate the load balancing behaviour of the file storage system under different conditions.

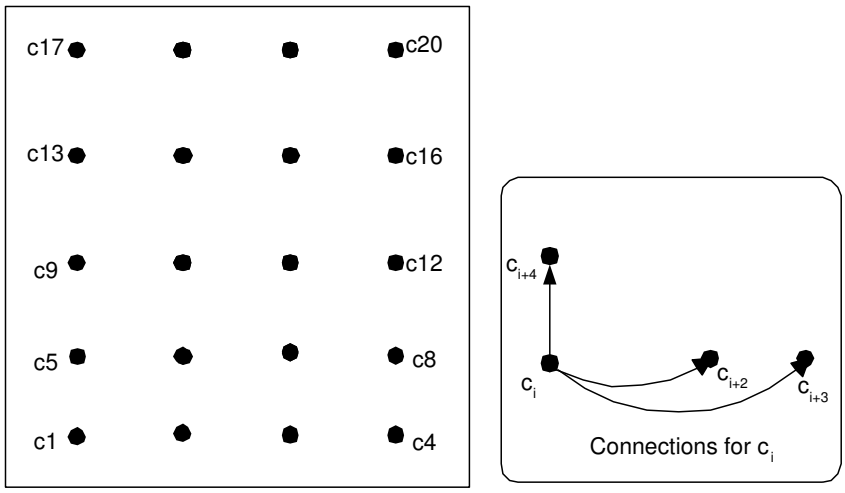


Figure 6.4: Component topology used in experiments.

In the experiments, the choice of topology for the file storage system is important as it is a potential source of variability and bias in experimental results. An important requirement for the experiments is to ensure that no global state is used, explicitly or implicitly, by the components in the system, and the topology should be completely decentralised. Given these requirements, the topology chosen for the experiments is a uniform grid topology with 20 K-Components (see figure 6.4), where each component has an incoming and outgoing connectivity degree of three. The maximum network distance (in terms of number of hops) is 5. This uniform topology is preferable to a random topology as the experiments are easily reproducible and it is not untypical of configurations found in grid computing environments (Nowicki et al., 2004). Scale-free and adaptive topologies were also possible configurations, but were not chosen as scale-free topologies have implicit structural knowledge of node capabilities.

In the uniform topology, each node is implemented as a single file storage component that is deployed in its own K-Component. A simple discovery service is used by components to lazily bind connectors to other components and generate the fixed network topology. The experiments are all based on the following topology:

- A group of 20 K-Components with local persistent storage for P fixed-size files and an unlimited in-memory buffer. The provision of a buffer of unlimited size enables clients to set a high rate for file arrivals, although a limit on the buffer size is that it should not exceed the memory capacity of the hardware used in the experimental setup.

- Every component, $c_i : i = 1..20$, in the topology has outgoing connectors bound to the components at positions $\{i+2,\ i+3,\ i+4\}$. For example, in figure 6.4 c_1 is connected to components $c_{3..5}$. Components from $c_{17..20}$ have wrap-around connections to components from $c_{1..4}$. There is a total of 60 connectors in the topology.

- The system has a set of Y load generating clients.

- λ_S is the client file storage rate that is set to 1500 milliseconds for experiments 1 and 2 and 500 milliseconds for subsequent experiments. The goal is not to approximate a file arrival rate for a real-life file storage system, but rather to simulate the operation of a generic file storage system that stores constant bit-rate streaming traffic from a fixed small number of sources.

- λ_R is the client file retrieval rate that is set to 200 milliseconds for all experiments.

6.3.1 Hardware and Software Configuration

For the experiments, the following configuration of software and hardware was used:

- A client machine: 1 GHz Pentium III processor, 512 MB RAM, Windows XP. The load generating client(s) were executed on the client machine.
- A server machine: 3 GHz Pentium IV, 1 GB RAM, Windows XP. Both machines are connected to each other via a 100 Mbit/second Ethernet. The average ping response time measured by Windows XP is 0 milliseconds, making it negligible for the results presented. The 20 K-Components were all executed on the server machine, as this enabled use of the system clock for timing events in the experiments.
- The system clock for the server machine, `time_t` from the standard C runtime library, is used to generate time information for experimental events. In win32 programs based on `time_t`, the margin of error of the system timer can be up to 10-15 milliseconds (DiLascia, June, 2004), which accounts for some of the minor oscillations in the timing patterns in the experiments but is considered acceptable for the purpose of the experiments in this thesis.
- K-Components, including Orbacus 4.1.1 (Concepts, 2001) for C++, JTC 2.01 (Concepts, 2000) and Xerces-C++ 2.4 (Project, Dec 2003).

6.3.2 CRL Parameter Tuning

There is a set of parameters and functions in CRL that can be tuned to affect the performance of the system in a given environment. The wide range of different possible configurations for the parameters prevents a more thorough evaluation of the effect of tuning the parameters in this thesis, where the concern is to demonstrate that CRL can establish system-wide properties without recourse to global state. The set of configurable parameters includes:

- Advertisement implementation strategy. Alternative approaches to advertising load costs include (1) RPC, (2) notification, (3) a hybrid RPC+Notification approach. For the experiments presented here, an RPC advertisement function was provided as there is a large amount of traffic in the system and a reduced need for asynchronous notification of load changes. RPC advertisements are supplied as return values to `store` and `forward` actions.
- Rate of Decay. The rate of decay, ρ, and unit time td, seconds in the experiments in this thesis, are configured to match expected changes in component availability. For dynamic networks with a large amount of traffic available to sample, cached V-values can be decayed more rapidly than in stable networks where there are less frequent events used to update the state transition model (see equation 6.1).
- Temperature. Increasing the value for the temperature T in Boltzman-Action selection increases the frequency with which sub-optimal load balancing actions are taken by adaptation contracts.

While this can produce sub-optimal system behaviour, it has the benefit of allowing adaptation contracts to discover changes in component load levels.

- Connection Cost Model. A static, configurable cost model is used in the experiments, but in decentralised systems such as MANETs a dynamic connection cost function that uses system services to calculate estimated connection costs would provide more accurate information about action costs to the CRL system.

- State Transition Model. The state transition model is a simple model of the probability of a *store* or $forward_i$ action succeeding based on sampling previous executions of the actions.

- A maximum cost, $MaxCost$, is defined for all experiments. Its value is set to -80.

- MDP Termination Cost Model. The reward model for a successful *store* action is designed by the component provider as a function $Load(c_i,\ f_{load}) \rightarrow \{Low, Med, High, Peak\} \in \mathbb{R}$. In the experiments, the following ranges are assigned to the different load levels, where $MaxLoad$ is the storage capacity of the component:

$$Low < MaxLoad/4, \quad Med <= MaxLoad/2, \quad High <= MaxLoad \times 3/4 < Peak$$

6.3.3 Experiment 1: Balance Load Over Homogeneous Components

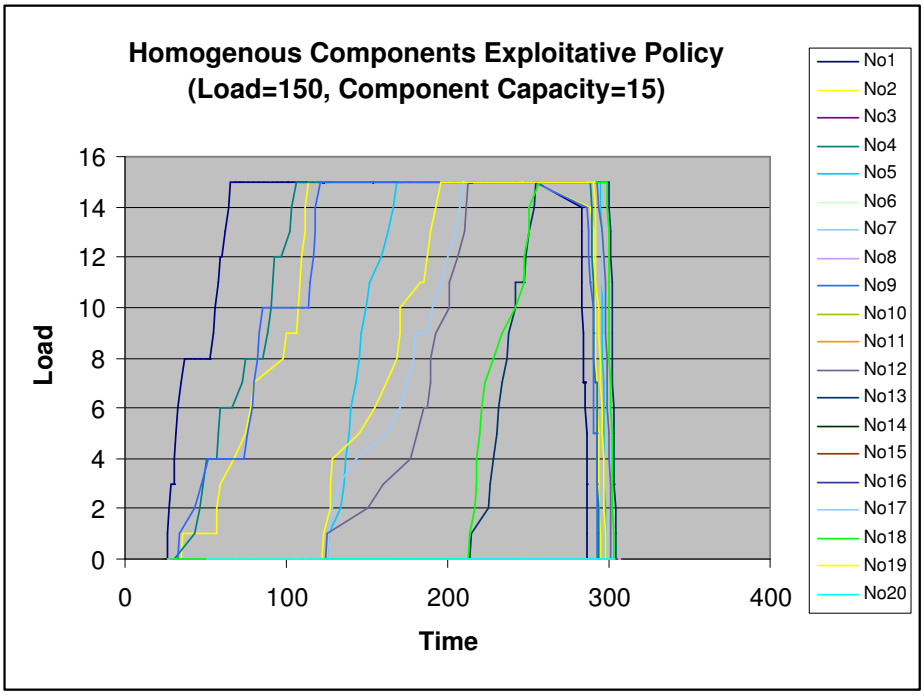
Storage Capacity	ρ	r_S	r_F	r_C	T	Low	Med	$High$	$Peak$
15	1.05	-1	-10	-2	2.5	-2	-3	-8	-15

Table 6.3: Experiment 1, 2: Homogeneous Component Experimental Settings.

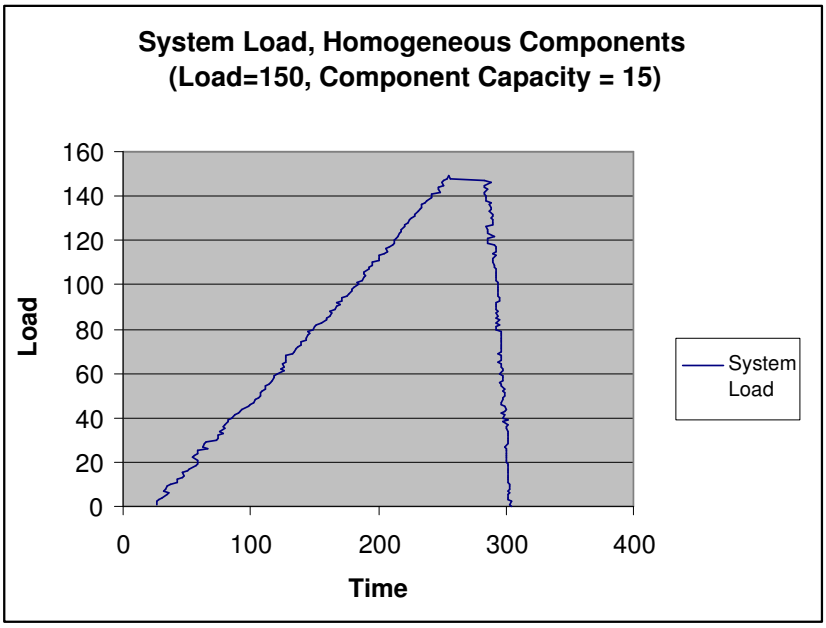
The goal of this experiment is to show how components can establish the system property of balancing a peak load among a group of components to make optimal use of (an unknown quantity of) system resources. In this experiment, a group of homogeneous components, with equal storage capacity of 15, are connected according to the topology in figure 6.4 and configured according to the settings in table 6.3. A single (peak) load generator sends file storage requests to component c_1 at a file arrival rate, $\lambda = 1500\ ms$.

The values for $Load(c_i,\ f_{load}) \rightarrow \mathbb{R}$, and r_C, are configured to ensure a high probability of storage actions being taken, relative to forwarding actions, when a file is received by a component. These values were: $Low = -2$, $Med = -3$, $High = -8$, $Peak = -15$, and $r_C = -2$. The connection costs are set to $r_S = -1$ and $r_F = -10$. Cached Q-values for $forward$ actions decay at a relatively high rate, with $\rho = 1.05$. The temperature parameter, T, is set to 2.5 to produce a greedy action selection policy, where more optimal actions are selected with higher probability.

As can be seen from the system load levels in figure 6.5, CRL balances resource usage among the group of components in the system. In fact, only 10 of the 20 components store the 150 files sent to the system, and *store* actions are always favoured over $forward$ actions by contracts when a file arrives in a component's buffer and the component has spare local storage space. This is because even though the reward model may indicate $forward$ actions should be favoured when a component's load becomes high, i.e., when a *store* action's reward is $Load(c_i,\ f_{load}) \rightarrow Peak$, the cached Q-values for $forward$



(a) Component Load Levels for Homegenous Model



(b) System Load Levels for Homegenous Model

Figure 6.5: Experiment 1.
Component and system load levels for homogeneous components with capacity=10 with a client generating a peak load of 150, with $\lambda = 1500\ ms$.

actions at state K also decay quickly over time while store actions are being executed, resulting in a lower probability of $forward$ actions being taken. When a component's local storage space is full, however, the contract forwards the file to a connected component with the lowest estimated load. Components at positions 3, 6, 8, 10, 11, 14, 15, 16, 17 and 19 do not receive any requests to store load and are unused in the experiment.

Also, minor oscillations in the system load can be seen in the system load graph in figure 6.5 (and in the other system load graphs in this chapter). This is due in part to the minor timing errors in win32, and contention between components for access to a shared log file. The unit of time used in all experiments in section 6.3 is seconds.

6.3.4 Experiment 2: Adapt Load Balancing Behaviour to Exploit the Introduction of a File Server with Increased Load Capacity

Storage Capacity	ρ	r_S	r_F	r_C	T	Low	Med	$High$	$Peak$
200	1.05	-1	-10	-2	2.5	-2	-3	-8	-15

Table 6.4: Experiment 2: File Server Component Settings.

The goal of this experiment is to show how components can adapt and learn to exploit the introduction of a file server with increased storage capacity. The homogeneous components are configured again according to the settings in table 6.3, with storage capacity of 15 units, and a single file server with increased storage capacity (200 units) is placed at component position c_{20} in the grid. The total load sent by a single load generator that interacts with the component c_1, and at a file arrival rate $\lambda = 1500\ ms$, is increased to 200 units.

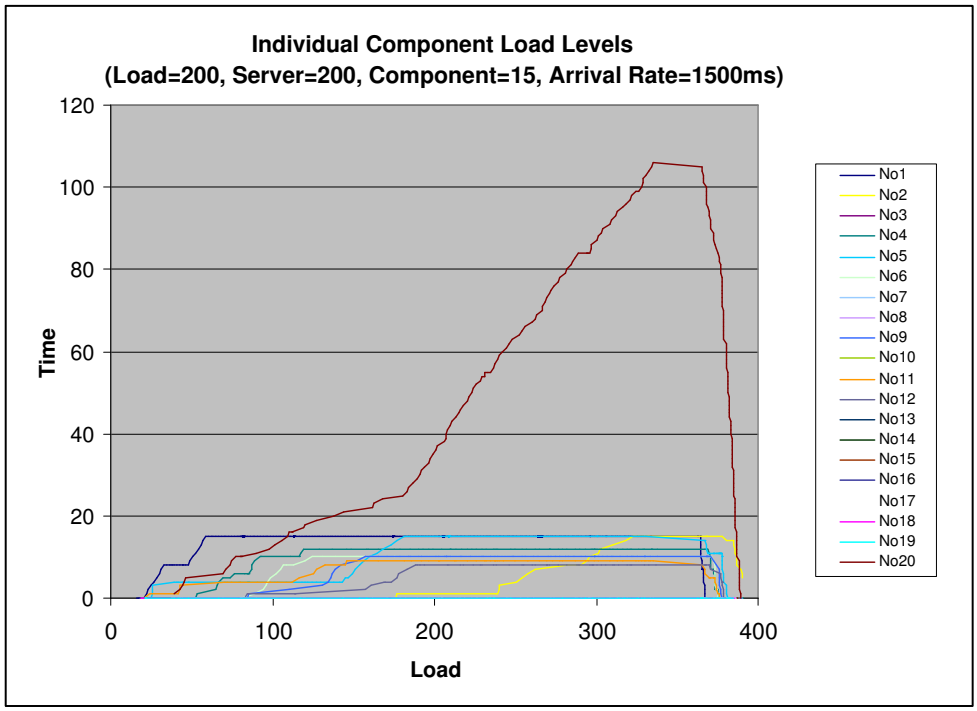
The reward model for *store* actions were set to $Low = -2$, $Med = -3$, $High = -8$, $Peak = -15$, and $r_C = -2$. The connection costs are set to $r_S = -1$ and $r_F = -10$. Cached Q-values for $forward$ actions decay at a rate of $\rho = 1.05$. The temperature T is again set to 2.5.

As can be seen from figure 6.6, load is again balanced over the components in the system and the use of system resources is exploited, including the newly introduced file server. This experiment shows how adaptation contracts are able to learn to exploit the superior storage capabilities of the single file server in the system. The objectives of adapting and optimising adaptation contract policies to a changed environment are met.

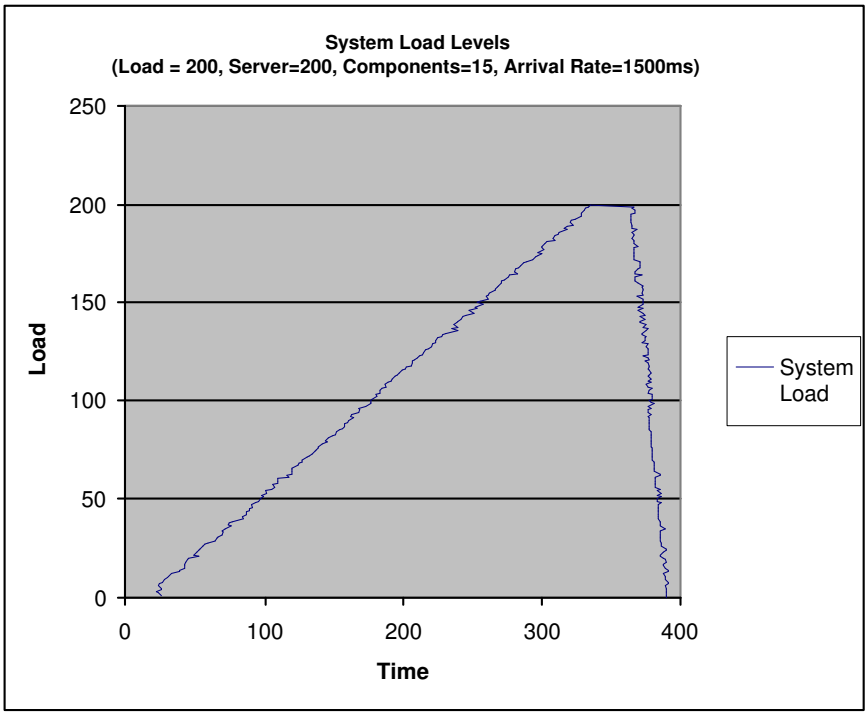
6.3.5 Experiment 3: Adapt the CRL Parameters to Optimise Load Balancing Behaviour for the File Server Scenario

Storage Capacity	ρ	r_S	r_F	r_C	T	Low	Med	$High$	$Peak$
10	1.05	-1	-10	-2	2.5	-2	-3	-8	-15

Table 6.5: Experiments 3 to 6: Homogeneous Component Settings.



(a) Component Load Levels for Server Model



(b) System Load Levels for Server Model

Figure 6.6: Experiment 2.
Component and system load levels for single server with capacity=200, homogeneous component capacity = 15. Total load from single generator=200. $\lambda = 1500\ ms$.

Storage Capacity	ρ	r_S	r_F	r_C	T	Low	Med	$High$	$Peak$
100	1.05	-2	-7	-3	2.5	0	-1	-3	-5

Table 6.6: Experiments 3 to 6: File Server Component Settings.

The goal of this experiment is to show how tuning the CRL parameters and the reward model parameters for the file server (see table 6.6) optimises the system behaviour of the load balancing application to more quickly exploit the single file server component. Similar to experiment 2, a single file server, but with lower file storage capacity (100 units), is placed at component position c_{20}. The settings for the homogeneous components are left unchanged from the values in experiments 1 and 2, but their storage capacity is reduced to 10 (see table 6.5). Again, a single load generator interacts with the component c_1, but in this experiment at a faster file arrival rate, $\lambda = 500\ ms$, and, similar to experiment 2, it generates a total load of 200. The file arrival rate is increased to demonstrate the scalability of the K-Component model to increased traffic levels. Connection costs and decay are set to the same values as in experiments 1 and 2.

The reward function for the *store* action in the file server component, $Load(c_i,\ f_{load}) \rightarrow Peak$, is tuned to increase the probability of files being stored by the file server component. The values were set to $Low = 0$, $Med = -1$, $High = -3$, $Peak = -5$, and $r_C = -3$. The connection costs for the file server component were set to $r_S = -2$ and $r_F = -7$ to increase the cost of forwarding a file.

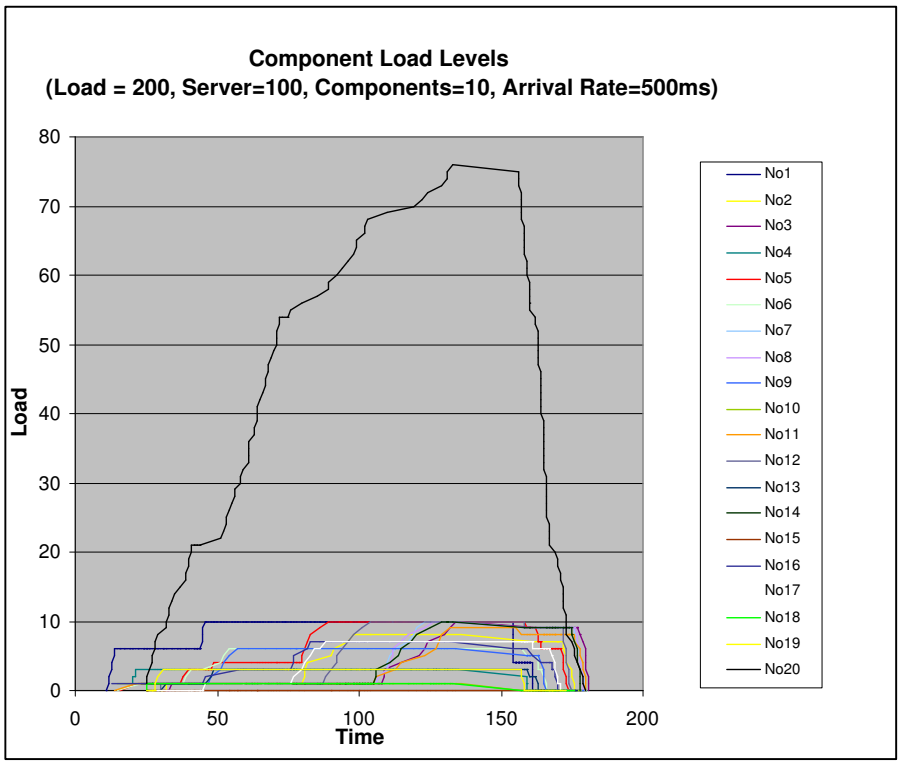
As can be seen from the results in figure 6.7 (where the scale changes due to the increased arrival rate), the system displays improved adaptability in learning to exploit the file server at position c_{20}. This experiment demonstrates how the CRL and reward model parameters can be tuned to optimise system performance and properties.

6.3.6 Experiment 4: Adapt System Load Balancing Behaviour to Exploit Two Storage Servers in the System

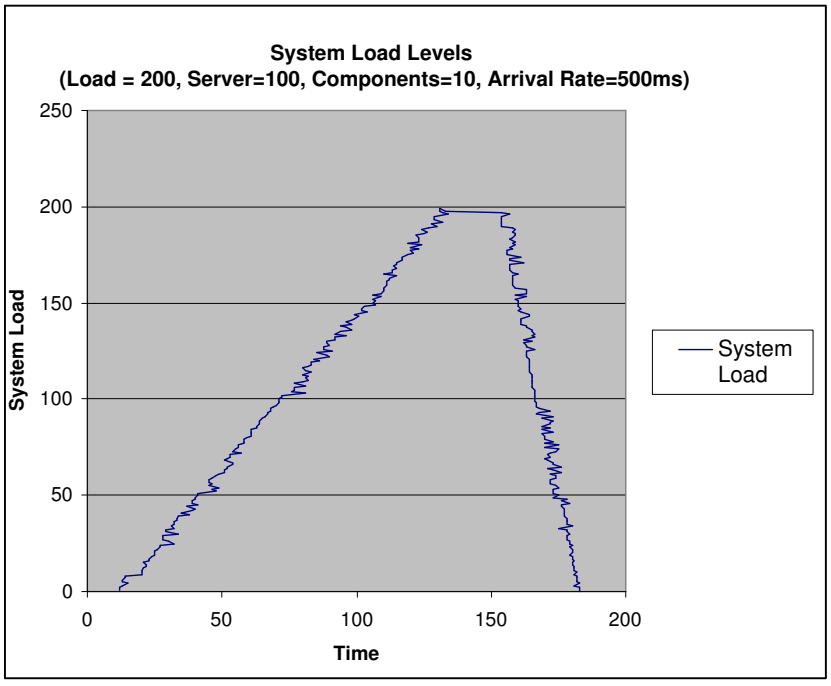
The goal of this experiment is to show how the system can adapt its behaviour to exploit the introduction of a second file server component in the topology. In this experiment, two servers with file storage capacity (100 units) are placed at component positions c_6 and c_{20}. The file server component settings are configured to the same values as those used in experiment 3 (see table 6.6), and the homogeneous components have the same settings as in experiment 3 (see table 6.5). Again, a single load generator interacts with the component c_1, generating an increased total load of 250, and as can be seen in figure 6.8, the system learns to exploit the superior storage capabilities of both file servers. This demonstrates that the system can automatically adapt and optimise its behaviour to a change in its environment.

6.3.7 Experiment 5: Exploitation of a Single File Server by Three Load Generators

The goal of this experiment is to show how the system continues to meet its system optimisation goals of maximal resource usage and load equalisation for a more linear load distribution. In this experiment, the number of client load generators is increased to three to produce a more linear load

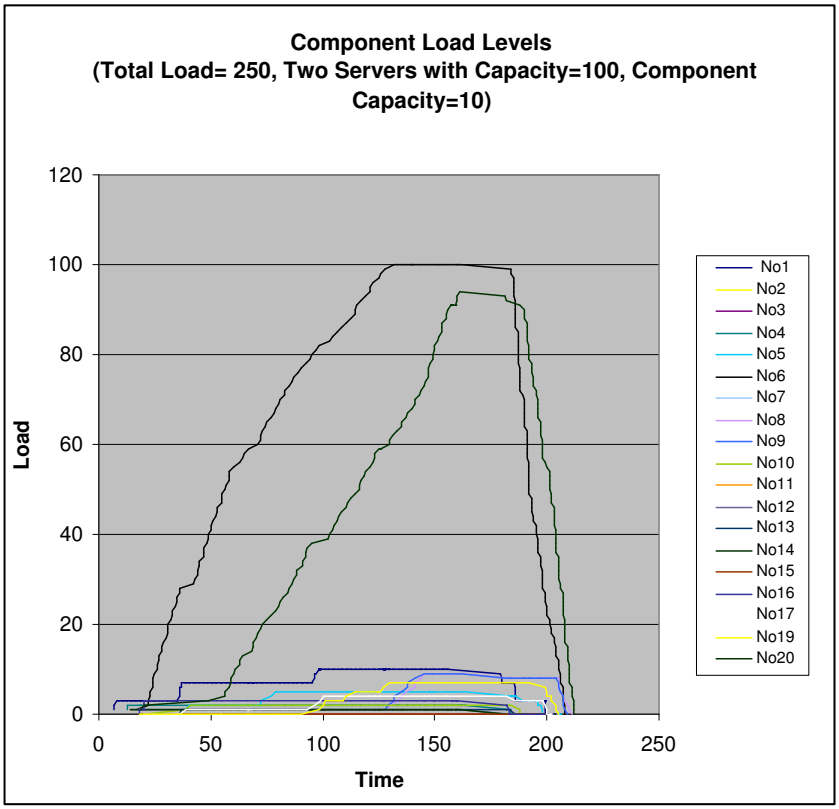


(a) Component Load Levels for Optimised Server Model

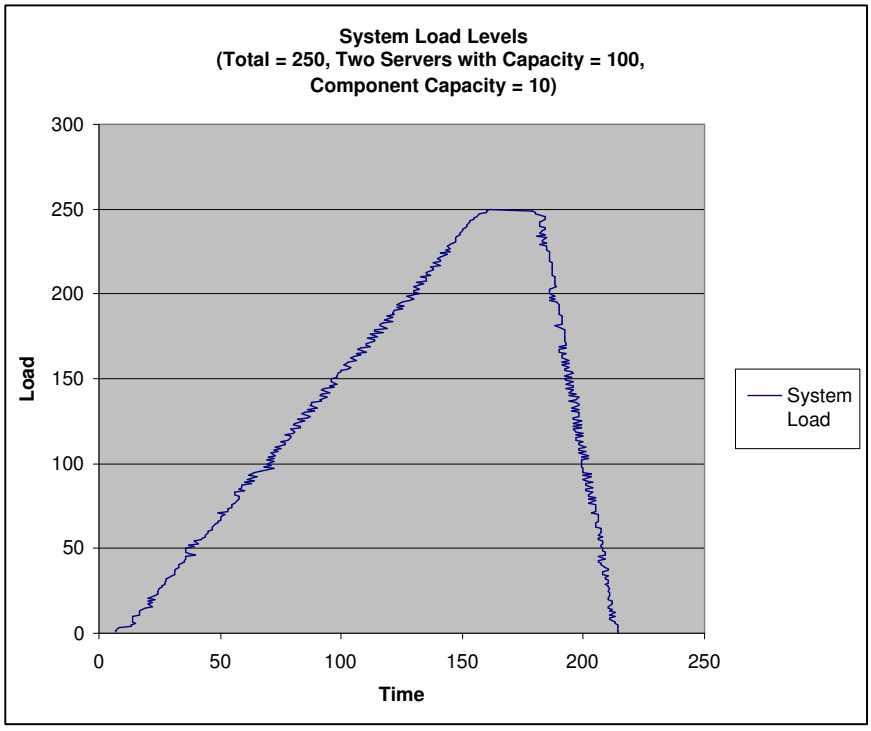


(b) System Load Level for Optimised Server Model

Figure 6.7: Experiment 3.
Component and system load levels for single server component with capacity=100, homogeneous component capacity = 10. Total load from single generator=200. $\lambda = 500\ ms$.



(a) Component Load Levels for Two Server Model



(b) System Load Level for Two Server Model

Figure 6.8: Experiment 4.
Component and system load levels for two file server components with capacity=100, homogeneous component capacity = 10. Total load from single generator=250. $\lambda = 500\ ms$.

model. The three clients send 80 file storage requests each, with a total load of 240, to the components at positions c_1, c_2 and c_3. The experimental settings are the same as in experiment 3 (see tables 6.6 and 6.5), with the single file server at c_{20}. As can be seen in figure 6.9, the system is able to adapt its load balancing behaviour to forward most requests from all three load generators to the file server and exploit its superior storage capacity. This demonstrates that the system properties continue to be maintained with increasing numbers of load generators.

6.3.8 Experiment 6: Self-Adaptive Load Generator that Discovers and Exploits a Server External to the System

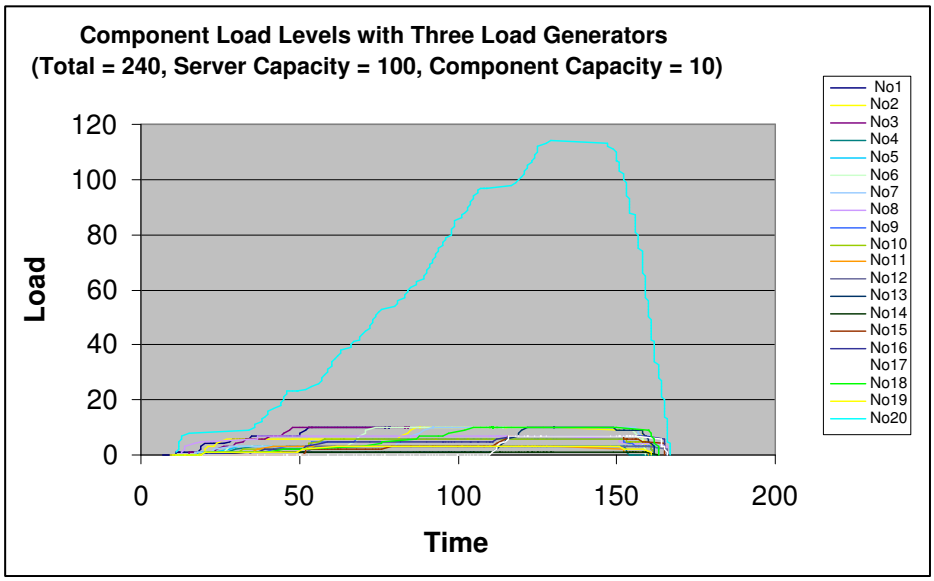
The goal of this experiment is to show how a load generator developed as a K-Component can use reconfigurable connectors and a component discovery service to exploit a component external to the load balancing system when the load generator's original connector to the system (at component c_1) is broken. The discovered component then binds to existing components in the system and balances its load with the existing components in the system. The configuration of components (single file server at c_{20}) and the experimental settings are the same as in experiment 3 (see tables 6.6 and 6.5)

```
outgoing storage (FileStorage::File::c_n1) {
  connector c1(FileStorage::File::c_n1);
  if (poll_state(c1,status)==CONNECTOR_BROKEN) {
     component S2 = discover(FileStorage);
     rebind_connector(c1, S2);
     jitter(10000);
  }
}
```

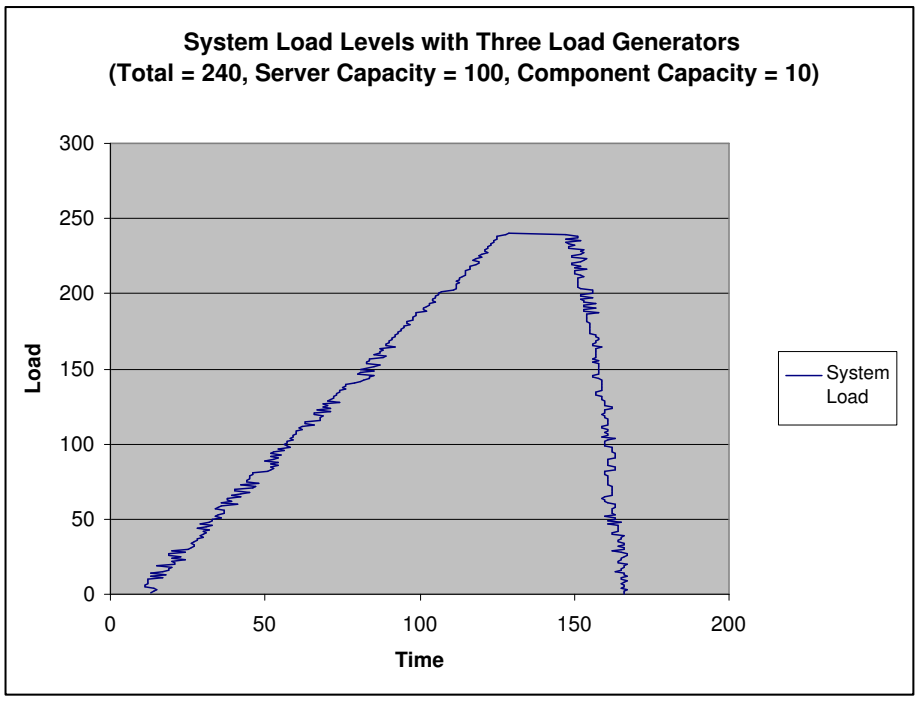
Table 6.7: Rule-based policy for a self-reconfiguring connector to a FileStorage component.

During the course of this experiment, a single load generator sends files to the component c_1, but at time t=128, the connection to c_1 is broken. The load generator contains an adaptation contract (see table 6.7) that identifies the broken connection, discovers a new component c_{21} (capacity=100), rebinds its connector to the new file server component and starts sending files to c_{21}. The new component c_{21} then binds its connectors to components $c_{3..5}$ and starts storing files locally from time t=140. It stores most of the subsequent files received locally, but several are forwarded to other components, including c_1 and c_{20}.

While the connector was being reconfigured, a total of nineteen files that the load generator would have attempted to send to a connected component were not sent. During connector reconfiguration, the load generator connector is blocked waiting for reconfiguration and as nineteen files were not dropped, this indicates that component discovery and connector reconfiguration lasted approximately 9.5 seconds ($19 \times 500ms = 9.5 \quad seconds$). A limitation of the implementation of the file storage system exposed by this experiment is that many of the files were not retrieved after connector reconfiguration as c_{21} receives requests to retrieve files from the system, but does not have forward references to files sent originally to c_1.

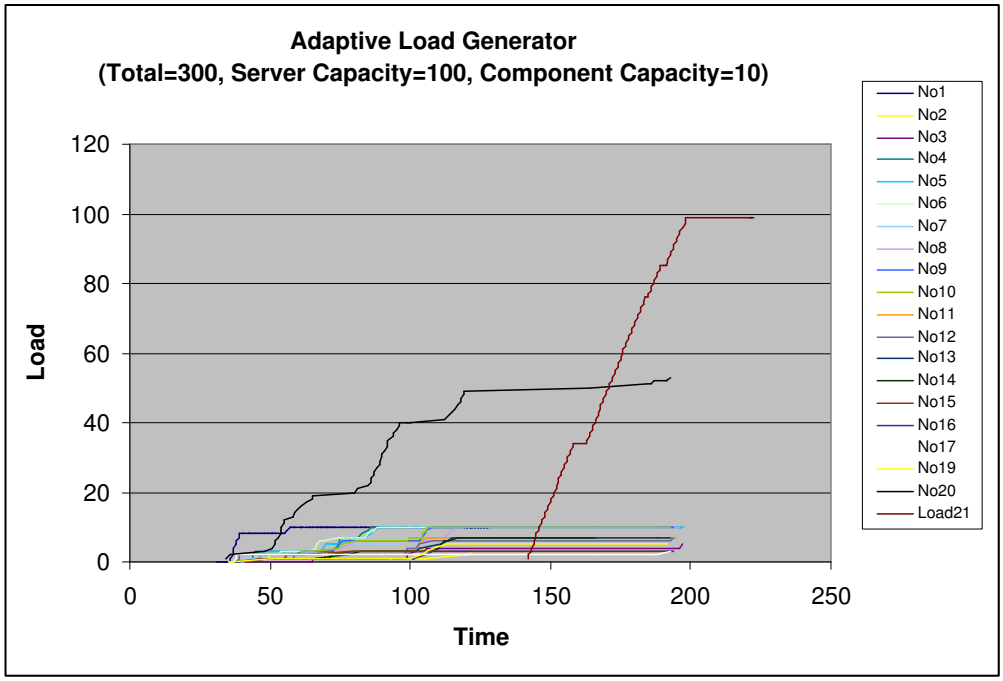


(a) Component Load Levels for Three Client Model

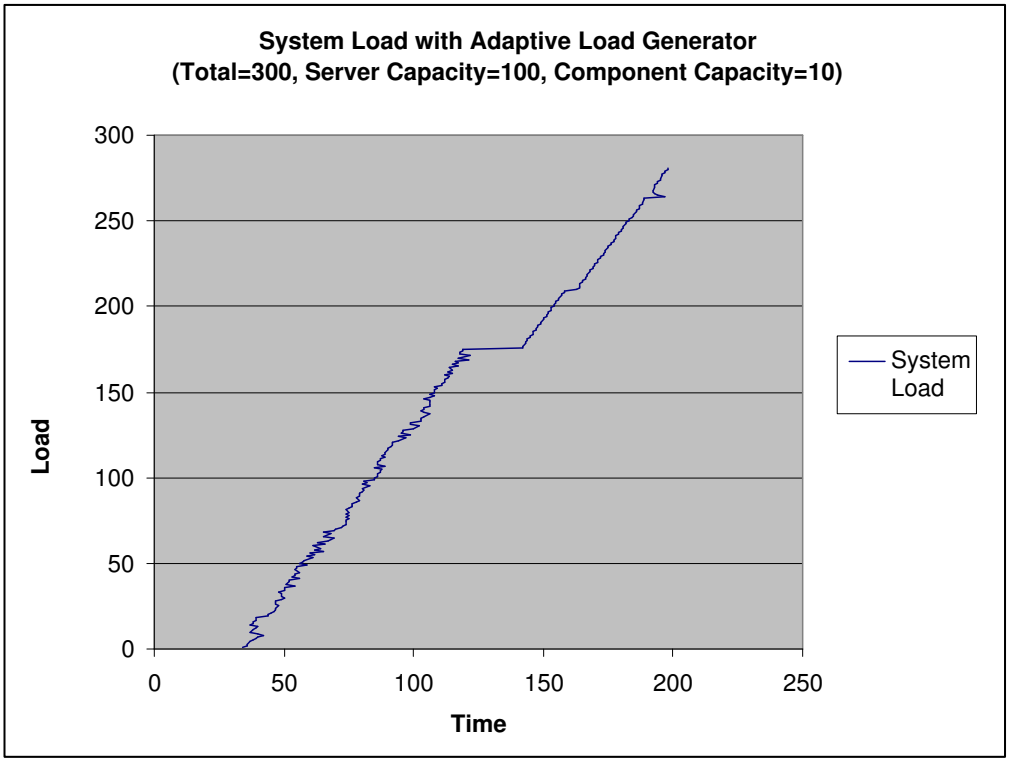


(b) System Load Levels for Three Client Model

Figure 6.9: Experiment 5.
Three clients generating a dispersed load to c_1, c_2 and c_3. Component and system load levels, single server component with capacity=100, homogeneous component capacity = 10. Total load from single generator=240. $\lambda = 500\ ms$.



(a) Component Load Levels for Adaptive Load Generator



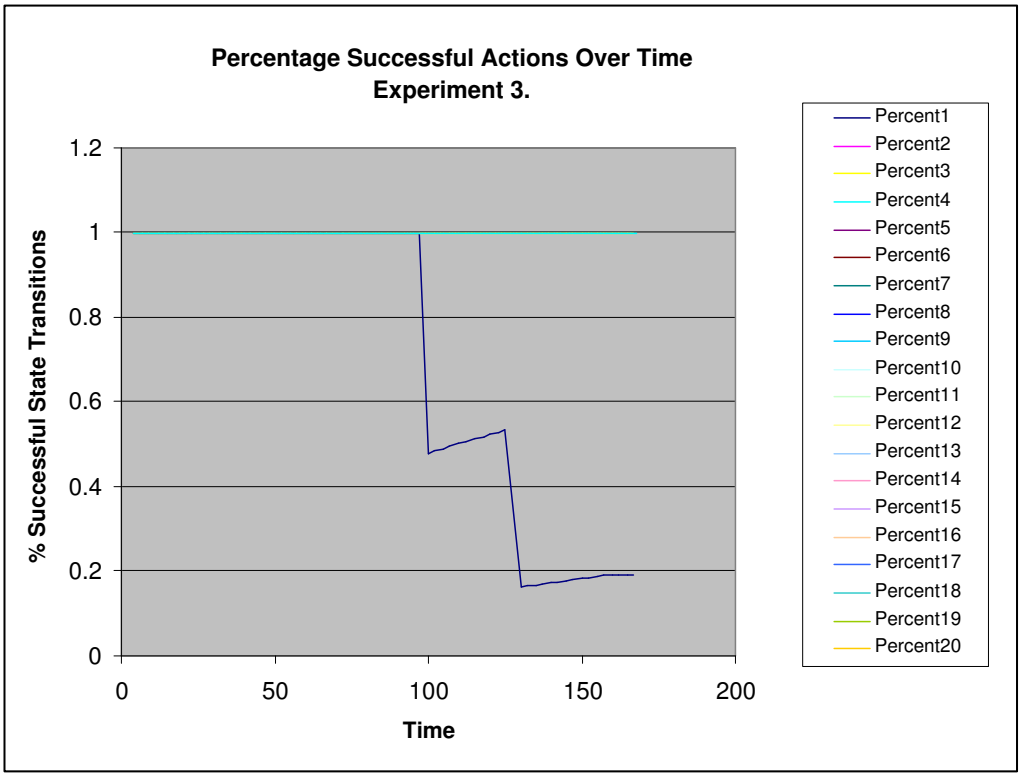
(b) System Load Levels for Adaptive Load Generator

Figure 6.10: Experiment 6.
Single, adaptive load generator identifies broken connection to c_1 at time t=120, discovers and binds to new component c_{21} (capacity=100) and starts sending files to c_{21} at time t=140. Server component c_{20} has capacity=100, homogeneous component capacity = 10. Total load from single generator=300. $\lambda = 500\ ms$.

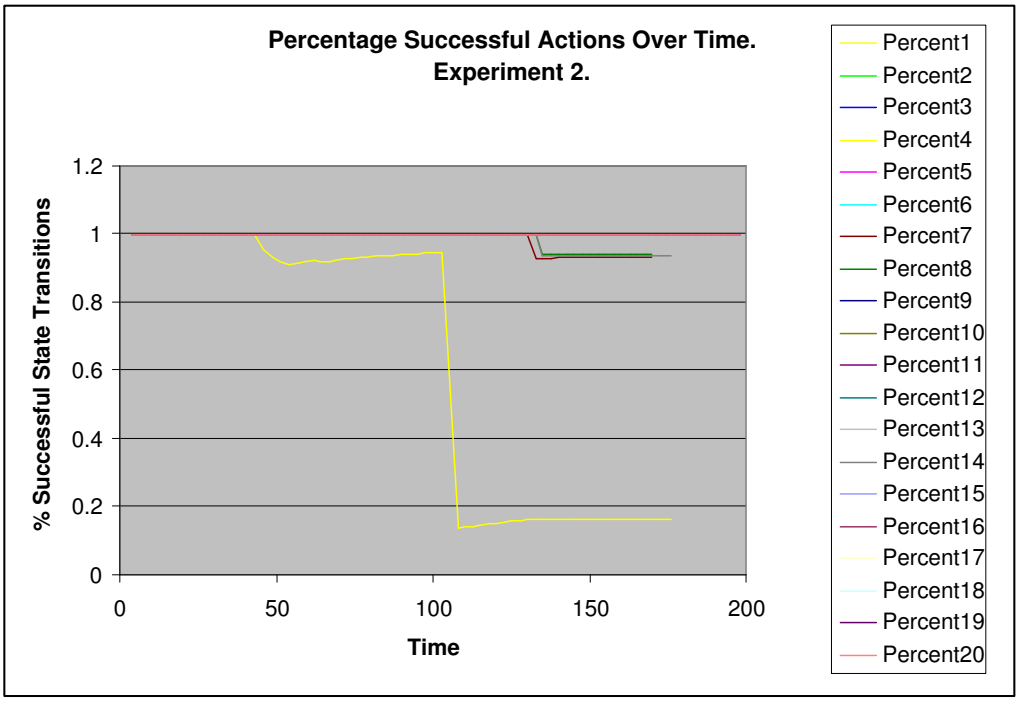
6.3.9 Other Optimisation Criteria

Another system property of the load balancing system is the number of times a file is forwarded before it is stored at a component. The file storage system should try to minimise this value. Table 6.8 shows the total number of times files were forwarded in each of the experiments and the ratio of successful forward/store actions. The load balancing system is not optimal for experiment 1 as it doesn't minimise the total number of messages required to balance the load in the system. For experiments 2 and 3, where the load is received at c_1 and mostly stored at c_{20}, a network distance of 5 hops (see figure 6.4) the forward/store action ratio is considerably improved. For experiment 3 the forward/store ratio produces only .25 more forward actions than the optimal 5 forwards/store if all files were stored in the server at c_{20}. For the two-server experiment, where a server is placed at c_6 at a network distance of two hops from the source of the load c_1, the forward/store ratio improves to a ratio of 4.75. For experiment 6, the forward/store ratio is considerably lower at 2.6933, as the load generator reconfigures to connect directly to file server c_{21}, where 100 of the 300 files sent to the system are stored without producing any forward actions.

The load balancing system can also be evaluated by its performance at optimising properties at the component-level instead of the system level. The percentage of attempted action executions (*store* or $forward_{0..2}$) performed by each component in experiments 2 and 3 are illustrated in figure 6.11. In experiment 2, only component c_1 shows a deterioration in the percentage of successful action executions performed, as all other components successfully execute every *store* or $forward$ action they attempt. However component c_1 after time 100 repeatedly and unsuccessfully attempts to execute *store* actions that fail as it has no spare storage space. The reason why c_1 tends to prefer *store* actions over $forward$ actions (that will ultimately successfully forward the component to the file server at c_{20}) is that the relative strength of the advertisements by c_{20} degrade over the 5 hops from c_{20}. At each hop, an additional connection cost of $r_F = -2$ and a MDP termination cost of $r_C = -3$ are added to the advertisement. The $MaxCost$ level for both *store* and $forward_{0..2}$ actions at c_1 is reached after time 100 and failed *store* actions do not update the policy ($MaxCost$ for the action has been reached), so they are repeatedly attempted and fail. The same reasons can be used to explain the increased failure of actions at c_1 from time 100 in experiment 2. However, setting the temperature T to 2.5 ensures that with some probability (20% in the case of experiment 2), component c_1 will attempt a $forward$ action. This way, load can continue to be balanced, even though component c_1 is not performing optimally.



(a) Experiment 3. Successful Actions Percentage.



(b) Experiment 2. Successful Actions Percentage.

Figure 6.11: Percentage of successful action executions performed by components in experiments 2 and 3.

	Store Requests by Load Generator	No. of Forwarded Requests (Cyclic)	Forwarded/Stored Ratio
Experiment 1	150	1050 (36)	7
Experiment 2	200	1361 (286)	6.805
Experiment 3	200	1048 (155)	5.24
Experiment 4	250	1188 (192)	4.752
Experiment 5	240	1227 (202)	5.1125
Experiment 6	300	808 (88)	2.6933

Table 6.8: Ratio of forward to store actions in experiments 1 to 6.

6.3.10 Feedback and System Properties using CRL

In the experiments presented in this chapter, the CRL agents, i.e., the adaptation contracts, learn to collectively balance load between connected components in the system using different feedback models, including evaluative feedback (Sutton and Barto, 1998) from the execution of load balancing actions, environmental feedback from the state transition models and collaborative feedback (Dowling et al., 2005) as advertised load costs. Negative feedback is also provided by the decay of cached load costs.

Evaluative feedback is received by CRL agents as rewards after executing `store` and `forward` actions. Agents use these rewards to update their policy using a distributed reinforcement learning algorithm. The state transition model helps agents learn an improved policy, i.e., learn the optimal load balancing action in the experiments, by building a stochastic model that provides environmental feedback as an estimate of the probability of an action succeeding.

In CRL an agent provides collaborative feedback to its neighbours through advertisement, e.g., by advertising a change in its load value. Advertisement in K-Components can be implemented using both RPC connectors and feedback events. The caching of a neighbour's advertised load costs reduces the amount of control traffic generated in the system, and in K-Components the cache is implemented as component feedback state values stored in the AMM. Cached load costs are used by the distributed reinforcement learning algorithm but these are only estimations of a neighbour's current cost. In the experiments, adaptation contracts made load balancing decisions based only on estimations of the value of loads of their neighbours, 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 forwarded to the wrong component due to stale cache information, but CRL helped improve the accuracy of cached estimates through both advertisement and decay of cached information. Advertisement also produced positive feedback in load balancing action selections in experiments 3 to 6. When actions were taken to balance files to the file servers, this increased the probability that other adaptation contracts would perform similar load balancing actions that sent load to a file server. This form of collaborative feedback helps improve the ability of the agents in the system to collectively learn and adapt to changes in the system.

The demonstrated ability of the file storage system to adapt its load balancing behaviour to changes in its environment, such as the appearance of a server with increased storage capability, is a function of the different feedback models in CRL. Negative feedback, in the form of decay, ensures that the probability agents of selecting actions learnt in the past will degrade over time if there is an absence of further actions or advertisement, while collaborative and environmental feedback help improve the

collective learning of policies in the system.

6.3.11 Reducing Uncertainty in Action Selection in Dynamic Environments

In the experiments, collaborative feedback helped reduce the amount of uncertainty in an agent's decision making by enabling neighbours to share their cached, partial view of the load in the system. Cascading advertisements help properties in the system, such as the presence of a file server, propagate further than their single hop neighbours. In the case where a group of agents collectively learn similar actions, such as forwarding files to the single file server in experiments 2, 3 and 5, we can say that the agents establish a weak form of emergent consensus on the optimal actions to perform in the system. The consensus is weak in that most, but not necessarily all, of the agents learn to perform actions that forward files to the file server. This system property can be clearly seen in the experimental results. The establishment of emergent consensus between a group of agents helps them overcome the uncertainty inherent in dynamic environments and can provide a mechanism for establishing collective, autonomic behaviour in decentralised systems, such as the collective forwarding of files to the file servers demonstrated in the experiments.

6.3.12 Assumptions of CRL

CRL makes several assumptions about decentralised environments that do not always hold for real systems. Firstly, there is an assumption that every agent in the system is trusted, as advertisements received from neighbours are used to update the agent's policy without examination of the source of advertisement to establish whether or not that source is a trusted sender of advertisements. CRL also makes assumptions about the semantics of its tuneable parameters that may not always hold true. In the load balancing experiments, each component has a common reward model for the execution of `store` and `forward` actions, based on their shared view of a component's storage capabilities. In effect, every agent in the load balancing system has a common reward model. CRL would be ineffective in the load balancing system if there was a lack of consensus by components on their reward models, or on how to calculate its local storage levels, or what constitutes a unit of storage space. In the case where there is no advance consensus between agents on the CRL model's parameters, agents would first have to establish consensus on the values (and possibly meaning) of each parameter, before they attempt to perform system optimisation.

6.4 K-Component Performance Testing

The experimental setup used for the experiments presented in this section is the same as in section 6.3.1. Experiments are performed on the FileStorage component defined in section 6.1 using adaptation contracts written using rule-based and ECA policies. The unit of time for the performance figures is milliseconds unless stated otherwise.

6.4.1 Experiment 7: Performance Comparison with CORBA

This experiment measures the performance of end-to-end K-Components invocations relative to the performance of the underlying CORBA transport protocol and provides a guide to the overhead introduced by asynchronous reflection during normal operation invocations. The benchmark tests performed are based on invocations of the `submit` operation provided by the `FileStorage` component, with a 1 Kb file passed as a parameter. The results from table 6.9 show that invocation times for operations using K-Component connectors introduce overhead of approximately two hundred percent to standard CORBA invocations based on Orbacus v4.1.1. The Orbacus round-trip invocation times are comparable with results from the Open CORBA Benchmarking paper (Tuma and Buble, 2001)[3]. There is no statistical difference between the average connector invocation times when adaptation contracts are installed and when they are not installed, indicating asynchronous reflection in K-Components introduces low overhead to normal component operation. In figure 6.12, however, the minimal overhead introduced by asynchronous reflection is visualised by the spikes in invocation time that occur periodically (for connectors with an adaptation contract) at invocation numbers 56, 110, 154, 208 and 242. These spikes are caused by context switches to and from the adaptation contract being evaluated and can be seen in the higher standard deviation of the invocation times for connectors with a contract. They correspond to the sampling time interval, t_c, that is set to 100 msec for all the experiments.

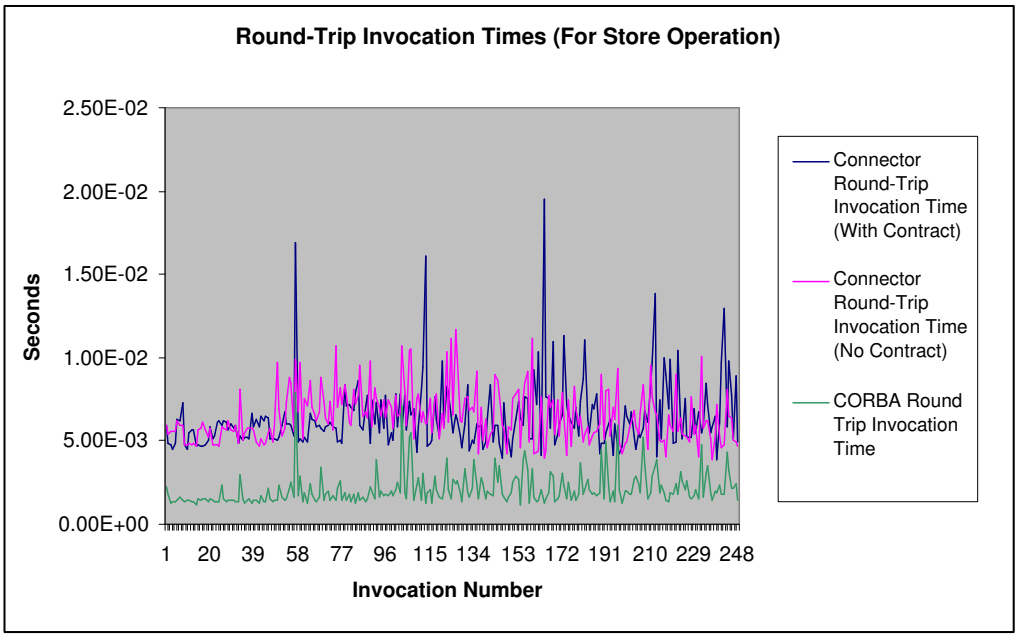


Figure 6.12: Experiment 7. Comparison of round-trip invocation times with CORBA.

[3]In the Open CORBA Benchmarking paper, a configuration using Orbacus 4.01, Linux 2.2.16, Intel Pentium 166 MHz, 64 MB RAM produced median round-trip times of 0.00027 seconds for static, null invocations. The increased overhead in the round-trip time for the `submit` operation can be accounted for by the marshalling/unmarshalling of the 1Kbyte file sent as a parameter.

	Connector Round-Trip Invocation Time (With Contract)	Connector Round-Trip Invocation Time (No Contract)	CORBA Round-Trip Invocation Time
Average Time	6.37	6.51	2.10
Maximum Time	19.47	16.00	7.88
Minimum Time	3.91	3.91	1.17
Standard Deviation	1.99	1.58	0.98

Table 6.9: Round-trip invocation times (in milliseconds). Performance comparison with CORBA.

	Broken Connector Identification	Rebind Connector	Adaptation Agility Time
Average Time	2430	56	2486
Maximum Time	3810	114	3924
Minimum Time	2060	10	2070

Table 6.10: Connector rebinding times using rule-based policy.

6.4.2 Experiment 8: Reconfiguring Connectors

The goal of this experiment is to demonstrate how a self-adaptive K-Component, with a single deployed FileStorage component of storage capacity 15, can reconfigure a connector to adapt to changes in the K-Component's internal state as well as changes in state external to the K-Component, (see figure 6.13).

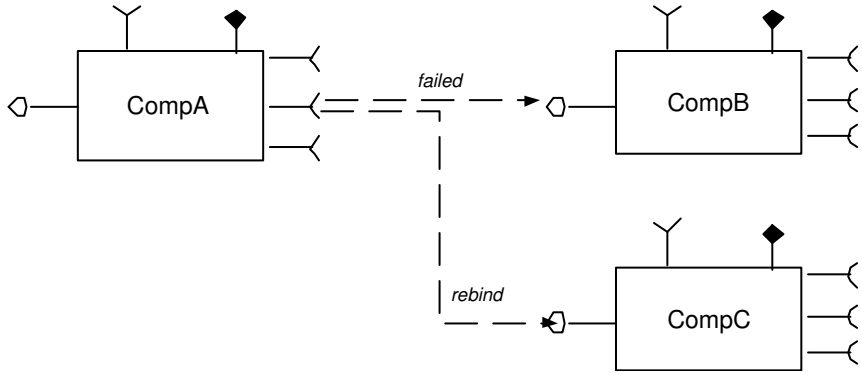


Figure 6.13: Self-healing connection.

Connector Reconfiguration on an Internal K-Component Adaptation Event

This experiment provides performance measurements for the self-healing connector described in experiment 6. The adaptation contract (see table 6.7) defines a rule-based policy that describes how to identify a fault in a connector, as a `CONNECTOR_BROKEN` status state on a connector object, and how to reconfigure the connector to use a different version of the connector's target component. The `discover` operation simulates the automatic discovery of an alternative target component. Performance figures for this contract are provided in table 6.10. A particular feature required by this adaptation contract is jitter control, in order to prevent the K-Component from attempting to repeatedly attempt to reconfigure the connection if reconfiguration fails. A total adaptation agility time can be defined for the self-healing connector as the sum of the time required to identify the broken connection, the average thread scheduling time, time taken to discover the new target component and the time taken

```
handler rebind_FileStorage {
   component S2 = discover(FileStorage);
   connector c1(FileStorage::File::c_n1);
   if (poll_state(c1,status)==CONNECTOR_BROKEN){
      rebind_connector(c1, S2);
   }
}
outgoing storageNotification (FileStorage::File::c_n1) {
    state load(FileStorage);
    predicate is_full(StateFull.xml);
    event adapt_high_load(load, is_full, Low, rebind_FileStorage);
}
```

Table 6.11: ECA policy that rebinds a connector when FileStorage is full.

```
<?xml version='1.0' encoding='utf-8' ?>
<dsg:cb-values xmlns:dsg='http://www.dsg.cs.tcd.ie'
                xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
                xsi:schemaLocation='http://www.dsg.cs.tcd.ie
            F:\\repository\\Schemas\\KEventDesc.xsd'>
        <dsg:unary-op dsg:op-name='12' dsg:value1='10'/>
</dsg:cb-values>
```

Table 6.12: Predicate descriptor for a FileStorage feedback event.

to rebind the connector.

Connector Reconfiguration on Feedback Events from an External K-Component

The goal of this experiment is to demonstrate how a self-adaptive K-Component, with a single deployed component, can register a remote feedback event containing the `is_full` predicate with a remote component, how the remote component determines when the `is_full` predicate is matched, notifies the feedback event to the client component, that then executes an associated `rebind_FileStorage` handler for the feedback event, (see table 6.11). The `rebind_FileStorage` handler reconfigures the connector to use a different target component. The performance measurements are provided in table 6.13. In this example the predicate for event notification (see table 6.12) is matched when the component's storage level increases above 10, since op-name '12' corresponds to the `greater-than` predicate. The feedback event notification time is measured from when the client component sends a `submit` request that causes the server component's load to increase above 10 to when the notification event received triggers the execution of the associated event handler.

The results show improved adaptation agility properties to the identification of the broken con-

	Feedback Event Notification Time	Rebind Connector	Adaptation Agility Time
Average Time	810	589	1399
Maximum Time	2670	1660	4330
Minimum Time	293	260	553

Table 6.13: Connector rebinding times using an ECA policy.

	Connector Binding	Proxy Load	Load Library	AddRef	Update AMM
Average Time	118	12.8	8.9	22.4	26.1
Maximum Time	148	13.6	9.4	96.4	29.4
Minimum Time	114	1.2	8.3	13.6	24.6

Table 6.14: Connector binding times (no contract).

	Connector Binding	Proxy Load	Load Library	AddRef	Update AMM
Average Time	109	24.8	19.8	19.5	24.5
Maximum Time	227	118.0	113	51.3	25.2
Minimum Time	66	13.6	8	13.9	23.1

Table 6.15: Connector binding times (with contract).

nector experiment. This is because the time taken to identify a broken connector after the attempted invocation of an operation on a connector is determined by the time taken by the Orbacus ORB to throw a *Transient* exception, i.e., the cause of the failed operation is unknown to the ORB (Henning and Vinoski, 1999). The *Transient* exception is caught by the ORB adapter proxy and is rethrown as a K-Component exception. The connector catches this exception, sets its status to *broken*, and rethrows the exception to the application level. This method of identifying and adapting to broken connectors is not ideal, as application programmers have to explicitly account for the fact that operations on connectors may fail and could succeed if they are retried, as an adaptation contract may repair them after the *Transient* exception is thrown.

6.4.3 Other K-Component Performance Measurements

A set of measurements were taken to perform more fine-grained investigation of the performance of connector binding and unbinding, as well as component loading and unloading in K-Components.

Connector Creation and Binding/Unbinding Time

The connection setup involves a number of steps, including the explicit dynamic loading of a component's library from disk, the creation of component and contract objects using entry points in the DLL and the starting of contract threads. The results obtained have been averaged over 10 experimental runs.

Connector binding to remote components involves a number of steps (see table 6.14), including the explicit dynamic loading of the stub to the remote component, provided as a shared library, from disk. The instantiation of the stub and contract objects from the shared library using entry points, accessed using the system call `GetProcByAddress`, and the starting of contract threads also make up part of the connector binding time. Finally, on binding to a remote component, a connector registers its feedback event manager IOR and feedback events using the `AddRef` operation.

Connector unbinding involves a connector calling `Release` on the `KBind` interface of the remote component, unloading its proxy and unloading the library for the proxy.

The results obtained in tables 6.14, 6.15, 6.16 and6.17 have been averaged over 10 experimental runs.

	Connector Unbinding	Unload Proxy	Release	Unload Library
Average Time	126	40	31.4	7.6
Maximum Time	572	248	99.7	7.9
Minimum Time	263	10	9.9	2.6

Table 6.16: Connector unbinding times (no contract).

	Connector Unbinding	Unload Proxy	Release	Unload Library
Average Time	166	77.1	2.34	0.872
Maximum Time	574	312.0	8.91	2.220
Minimum Time	19	7.3	1.06	0.615

Table 6.17: Connector unbinding times (with contract).

Component Loading/Unloading Time

For completeness, the component loading/unloading performance figures for component replacement are presented in table 6.18. In the experiments, the component was in a passive state prior to the execution of the component replacement operation. As can be seen by comparing the load library times in table 6.14 with the component load times, the performance of component loading/unloading is comparable with dynamic linked library loading/unloading times for Win32.

	Component Load	Component Unload
Average Time	65	73
Maximum Time	253	310
Minimum Time	11	5

Table 6.18: Component loading/unloading times.

6.5 Analysis of K-Components as Autonomic Components

As shown in experiment 8, a K-Component can automatically identify and reconfigure a broken connector as well as adapt itself in response to feedback events that describe a change in its external environment. These are just two examples of simple autonomic behaviours that can be explicitly programmed in K-Components. The automatic repair of broken connections is an example of a self-healing autonomic property for K-Components, while the adaptation of a K-Component in response to external feedback events can be used to self-optimise a component's operation. For example, a self-optimising component could define a feedback event as a predicate on some expected service level by a remote component and if the remote component cannot meet the expected service level, the notification of feedback events can be used to trigger the automatic reconfiguration of the client component to use a different component, if available, that meets its desired level of service.

Asynchronous reflection provides autonomy to decision policies in the K-Component model. It enables reflective programs to reason asynchronously about the component's operation using feedback states and adapt their structure using architectural reflection. It enables the pro-active adaptation of a component to a changing environment, orthogonal to computational activity in the component. This is an important characteristic for autonomic components, as it enables components to self-optimise and self-heal during idle processor time.

The different autonomic component behaviours that can be programmed using the K-Component model are constrained by the set of feedback states that can be monitored and the range of adaptation actions that can be performed on components and connectors. Feedback states and feedback events can be defined on components, however the only support for a system feedback state is the status state defined on connectors. While K-Components provides support for reasoning about the state of application-level components and the status of connectors, there is no support for lower-layer events concerning the underlying middleware or operating system. Such support could be provided by repackaging the lower-level software as K-Components. Autonomic components require complete and accurate information about their operation to guide their self-adaptive behaviour, and future autonomic components will require access to information about the operating state of lower-level system software. This can be achieved through the instrumentation of system software with component models such as K-Components.

K-Components supports two different types of adaptation actions that can be used to realise autonomic behaviours: architectural adaptation actions that can reconfigure the K-Component's architecture meta-model and component actions. Architectural adaptation actions are useful for reconfiguring connections to faulty or poorly performing components, while component actions can be used to perform fine-grained adaptation to the behaviour of a component, e.g., change the implementation strategy of some internal algorithm, or request the component to perform any action. Architectural adaptation actions make use of KOM to provide K-Components with the platform independent ability to load and unload, at runtime, components written in C++. The runtime evolution of components in an autonomic computing system requires such a facility to be able to evolve system behaviour, e.g., to meet changed user requirements or to adapt to a changed external environment. However, the K-Component's AMM is constrained to support only static interfaces, defined at component design time. There is no support for dynamic interfaces in the AMM, as they would introduce additional system integrity management problems to the system (Dowling and Cahill, 2001a).

6.5.1 K-Components and the Requirements for an Autonomic Component

In chapter 2, a set of features were identified that a self-adaptive component model should provide in order to enable the construction of autonomic distributed applications. These are a state model to describe runtime component operation, adaptation actions that can safely adapt the component, a decision making entity that encapsulates the component's self-adaptive behaviour as a decision policy, techniques to evaluate and update the decision policy over time, and a decentralised coordination model to support system-wide adaptation of groups of components. In K-Components, these requirements are met by providing:

1. User-defined states can be defined on a component in K-IDL and a connector status state is also supported to reason about connector operation. The values of these states can be monitored at runtime by an adaptation contract to reason about the operation of the component and its connectors, and possibly trigger its adaptation.

2. A set of architectural adaptation actions are supported that enable the reconfiguration of the K-Component by an adaptation contract at runtime. User-defined actions can also be defined

on a component in K-IDL to adapt component operation at runtime.

3. The integrity and consistency of architectural adaptation actions is guaranteed by providing a RPC-consistency reconfiguration protocol that ensures that connectors and components have reached a reconfiguration safe-state before adaptation at runtime. Component developers have to ensure that component actions maintain application integrity and are at a minimum thread-safe.

4. Adaptation contracts are decision making agents that encapsulate a decision policy for adapting a K-Component. The ACDL supports the declarative specification of decision policies and different types of decision policies supported include rule-based policies, ECA policies, RL policies and CRL policies.

5. Decision policies can be updated in two different ways in K-Components. Action policies that are encapsulated in adaptation contracts can be automatically loaded and unloaded at runtime when a contract is associated with a proxy to a remote component. When a connector binds to a remote component, the contract is loaded and started automatically and when the connector unbinds from the remote component the contract is unloaded. In the case of the learning decision policies, RL and CRL, their policies are evaluated and updated continuously during system operation.

6. CRL provides a distributed reinforcement learning algorithm that allows a component to co-ordinate changes in its self-adaptive behaviour with its connected components. CRL provides advertisements and decay as mechanisms that update a K-Component's partial view of a wider system, and delegation actions enable the coordination of solutions to decentralised problems, such as the load balancing application in this chapter.

6.6 ACDL as a Programming Language for Autonomic Components

The ACDL provides a declarative programming model for developers with which they can specify decision policies describing the self-adaptive behaviour of K-Components. The syntax is based on C++ syntax and, as such, should be familiar to existing CORBA/C++ developers. The following discussion covers the expressive power of the different policies that can be specified using the ACDL, as well as the distributed systems considerations when designing a policy and the available techniques for evaluating and updating the policies.

6.6.1 Action Policies

Rule-based action policies can be used to reason about and adapt local components and connectors in a K-Component. A rule-based action policy is encapsulated in an adaptation contract and scheduled to run periodically. It evaluates rules consisting of conditions on component or connector state and performs any associated actions if a rule is matched. Rules are specified as `if-then` statements in the

ACDL. Rule-based policies, however, are not suitable for reasoning about remote components due to the increased network traffic produced from periodic polling of remote components.

Event-condition-action policies are designed to support reasoning about remote, connected components, and the conditional adaptation of a K-Component based on the state of a remote component. Feedback events can be defined as predicates on the remote component's state and associated handlers that are executed if the predicate is matched. Feedback events are registered with remote components on binding, and if they are matched, the updated remote component state value(s) is notified to the source K-Component. This notification can result in the execution of an associated handler that can contain actions (or a rule policy) to adapt the K-Component. The ACDL also supports the specification of ECA policies based on local components in a K-Component.

Both rule-based and ECA policies have equal expressive power in that they can reason about the state of a K-Component using feedback states and perform conditional actions. Both policies, however, suffer from scalability problems. As the number of feedback states and actions available in a K-Component grows, it becomes increasingly difficult to specify policies that handle all possible internal and remote states or be able to accurately predict the outcome of executing some action, particularly in dynamic environments.

In K-Components, there is no support for the automated evaluation of how well action policies meet some high-level self-management objective. Similar to existing systems (see chapter 2) the evaluation of the performance of a K-Component's self-adaptive behaviour can be performed by a system administrator observing its operation. Action policies can be updated in a running system, by binding and unbinding connectors and updating their associated adaptation contracts. This requires implementing new adaptation contracts, and dynamically loading them into the system as shared libraries. To summarise, action policies are useful for designing lightweight, autonomic components that operate in known or static environments and require relatively stable self-adaptive behaviours.

6.6.2 Learning Policies

A RL policy can be used to learn to associate component feedback states with component actions or architectural adaptation actions in order to maximise reinforcements (rewards) supplied by the component. Component providers implement a reward model that supplies reinforcements based on the state of the component, the action executed and the outcome of the action, which is generally determined by observing state changes in the component. Similarly for architectural adaptation actions, a reward model must also be provided, although this requires modification to the K-Component framework, as architectural adaptation actions are provided by the ArchReflect MOP.

Learning policies are useful where the space of component states and adaptation actions is large, and also where the outcome of an adaptation action cannot be predicted by the adaptation contract designer. Reward models can be used as a basis for evaluating adaptation actions and used to update and learn a decision policy over time.

Automated Evaluation of Actions

Reward models are designed by component providers to provide evaluative feedback on adaptation actions, thus enabling a policy to automatically learn more optimal actions over time. Reinforcement learning in the ACDL provides such support for the unsupervised learning of a policy. Reward models can be based on simple models such as the success or failure of architectural adaptation actions or runtime measurements made by a component, e.g., a connector's reconfiguration performance or the current load at a component. Typically, a component programmer evaluates the outcome of actions at runtime by observing changes in a component's internal state or the state of its environment. It can then supply a reinforcement to the learning agent that is based on the observation. For example, in the `FileStorage` component, when a *store* or $forward_i$ action is performed on a component and the file is successfully stored or forwarded, the component returns a reinforcement that reflects the success or otherwise of the action. Learning policies are useful for uncertain environments where properties, such as the probability of action success, may not be known at design time, but may be adapted to at runtime.

CRL as a Decentralised Learning Policy

Where learning policies need to operate on state that is external to a K-Component, CRL can be used as it model's a K-Component's local view of the system as a cache and provides advertisement and decay for updating that local view. The cache is provided in K-Component as component feedback state values stored in the AMM. The cache provides a K-Component with an estimated model of the state of its immediate, connected components. In CRL, the distributed RL learning algorithm uses the cached values for remote component states, as well as the connection cost model and the MDP termination cost model to learn an optimal policy for some system optimisation problem. Decay of the cache provides negative feedback to a K-Component's local view of the system, and enables a learning policy to adapt to changes in its environment.

In general, learning policies are useful for designing autonomic components that operate in dynamic environments where the result of self-management actions cannot be known at design time and where there is inherent uncertainty when a component interacts with its environment. CRL provides abstractions that allow components to collectively learn self-management policies in decentralised environments, by providing a component with a local, partial view of the system with its cache, collaborative feedback for updating the cache through advertisement and decay for providing negative feedback on the local view. Delegation actions enable agents to coordinate the solution to distributed problems by transferring the responsibility for solving discrete optimisation problems between agents.

6.7 Comparison with Existing Systems

A feature-based comparison of K-Components is presented in tables 6.19, 6.20 and 6.21. As can be seen in table 6.19, K-Components provides similar capabilities to existing self-adaptive middleware systems such as QuO and OpenORB for reasoning about distributed component operation using models of component state, although it provides less support for reasoning about system state. This could

be provided, however, by re-engineering existing system software as K-Components. K-Components also provides support for architectural adaptation similar to functionality provided by Georgiadis' software architecture and OpenORB. This functionality guarantees system consistency through a RPC reconfiguration protocol.

	State Model	Adaptation Actions Supported	Adaptation Consistency
K-Components	AMM, Connector State	Intrusive, Non-Intrusive, Internal	√
QuO	Middleware, System State	Intrusive, Non-Intrusive, Internal	√
OpenORB	AMM, Middleware, System State	Intrusive, Non-Intrusive, Internal	√
Accord	Component State	Intrusive, Non-Intrusive, Internal, External	Not Specified
Georgiadis	Software Architecture State, Component State	Intrusive, Non-Intrusive, Internal	√
Khare	Application State using Estimator Functions	Non-Intrusive	N/A
Hinnelund	Component State	Non-Intrusive, Internal, External	Not Specified

Table 6.19: Comparison state models and adaptation actions.

	Centralised	Distributed Coordination	Decentralised Coordination
K-Components	√	√	√
QuO	√	X	X
OpenORB	Limited	X	X
Accord	√	X	X
Georgiadis	X	Local Arch. Constraints	X
Khare	X	X	Decentralised Arch. Style
Hinnelund	X	X	X

Table 6.20: Comparison of coordination models.

K-Components, however, allows the construction of more types of coordination models and decision policies than existing systems. In particular, K-Components is the only system that provides support for the construction of centralised, distributed and decentralised coordination models (see table 6.20). K-Components is also the only system that supports both action and learning policies to encapsulate a system's self-adaptive behaviour (see table 6.21). K-Components is also the only reflective system that supports executing a decision policy asynchronously to system operation. Finally, CRL is also the only decentralised learning policy supported by any of these systems.

6.8 Summary

This chapter described an evaluation of the work presented in this thesis. In section 6.1, the objectives of the evaluation of K-Components and CRL as building blocks for autonomic systems were outlined. Each objective introduced was investigated and evaluated using experiments on a decentralised, load-balancing file system implemented using K-Components and CRL, as well as self-adaptation experiments on a K-Component.

	Decision Policy (DP)	Decl-Prog	DP Exec.	DP Evaluation	DP Updates
K-Comps	Rule-Based, ECA, Learning Policy	√	Async	Component/ Contract	CRL
QuO	Rule-Based, ECA	√	Async/Sync	Sys-Admin, GUI	X
OpenORB	Rule-Based, ECA	X	Sync	Sys-Admin	Manager Component
Accord	Rule-Based	√	Async	Sys-Admin	Composition Agent
Georgiadis	Rule-Based, Arch-Constraints	X	Async	Sys-Admin	Selector Functions
Khare	Estimator Function	X	Async/Sync	X	X
Hinnelund	Learning Policy with Critic	X	Sync	Critic/Learner	Iterative Policy Search

Table 6.21: Comparison of decision policies.

The experiments were divided into different scenarios, focussing on different system properties of the load balancing application that emerge from locally specified behaviour at K-Components as well as the self-adaptive performance of K-Components. Some performance measurements for architectural events such as component loading and unloading as well as connector binding, unbinding and rebinding were presented. A performance comparison of K-Components with CORBA was used to investigate the extra overhead introduced by connectors and asynchronous reflection.

An analysis of the K-Component model as a model for building autonomic components is also presented. It includes an evaluation of the capabilities and limitations of the ACDL for defining autonomic component and system behaviour. Finally, a feature-based comparison of the K-Component model with existing systems shows how the K-Component model provides more extensive support than existing systems for developing self-adaptive components and decentralised system software.

Chapter 7

Conclusion

> "Where I am, I don't know, I'll never know, in the silence you don't know, you must go on, I can't go on, I'll go on."
>
> Samuel Beckett, The Unnameable (1958)

This thesis described the design and implementation of a self-adaptive component model called K-Components and showed how the decentralised coordination of self-adaptive components can establish and maintain autonomic properties in distributed systems that operate in dynamic and uncertain environments. CRL was introduced as a decentralised optimisation technique that can be used to build decentralised coordination models, and its ability to establish and maintain autonomic properties was evaluated using a load balancing application developed using K-Components and CRL.

In this chapter, the contents of this thesis are summarised, an overview of the objectives of the thesis are presented and how they were fulfilled, and the contribution of this thesis to the state of the art is outlined. To conclude, future research ideas for the work presented in this thesis are discussed.

7.1 Thesis Summary

Chapter 1 provides background for the autonomic computing paradigm and motivates the use of bottom-up techniques to engineer autonomic computing systems. The K-Component model and CRL are then introduced, and features of K-Components such as the component and connector models, architectural reflection, and asynchronous reflection are discussed. Then, background to CRL and the different feedback models it provides to tailor its learning properties to decentralised environments are presented. Finally, the objectives of the thesis are stated and a definition of an autonomic computing system is provided.

Chapter 2 starts with a description of the limitations of existing consensus-based approaches to building autonomic computing systems in decentralised environments, and then a set of requirements for a self-adaptive, autonomic system is developed. The related work in the field is then reviewed, in terms of the requirements presented for a self-adaptive, autonomic system. The review highlights the achievements and limitations of existing work in self-adaptive systems, as well as recent decentralised approaches to building distributed systems.

Chapter 3 describes the K-Component model the main concepts of the K-Component model, including the component model, connectors, architectural reflection, asynchronous reflection and the ACDL. An example of a self-adaptive component is provided and the main features of the ACDL discussed. Chapter 4 describes the CRL algorithm, including the different feedback models in CRL and how to specify a CRL policy in the ACDL.

Chapter 5 presents the implementation of the K-Component model. It describes the language mappings for K-IDL and the ACDL and the K-Component framework in detail. A description of how K-Components are built using CORBA is also provided. Chapter 6 evaluates both K-Components and CRL as building blocks for autonomic computing systems using a decentralised load balancing, file storage application. The application exhibits autonomic properties of adaptive, system optimisation to a changed environment and self-healing in the presence of broken network connections. Performance measurements for K-Components and a feature-based comparison of K-Components with the reviewed systems are also provided.

7.2 Contributions

This thesis addressed the problem of developing self-adaptive components that can coordinate their operation to exhibit autonomic properties in uncertain and dynamic environments. The main contributions of this thesis are the K-Component model and CRL. The thesis identifies self-adaptive components as a building block for autonomic components and collaborative reinforcement learning as a coordination technique for building decentralised systems with autonomic properties. The K-Component model contributes a programming model for building self-adaptive systems that includes a component model and the adaptation contract description language. Self-adaptive behaviour can be specified in the ACDL as rule-based, ECA and learning policies. A model of asynchronous reflection enables the execution of adaptation logic asynchronously to program execution, and the continuous monitoring of components. The model meets the six requirements for a self-adaptive autonomic computing system presented in Chapter 2, i.e., K-Components provides a state model for components, a set of architectural adaptation actions and component adaptation actions, mechanisms to ensure system integrity during architectural adaptation, adaptation contracts as autonomous decision making components, mechanisms to update an adaptation contract's decision policy over time and CRL as a decentralised coordination model to manage collective system adaptation behaviour.

A decentralised file storage application was developed using K-Components and CRL that can self-optimise to a changing environment without the use of global information, and self-heal by reconfiguring faulty connections at the component-level. The application shows how CRL can establish system-wide autonomic properties, such as the adaptation and optimisation of the system to a changed environment and near-optimally balanced load over a group of connected components, in dynamic and uncertain environments.

7.3 Future Work

The scope of the different domains covered by this thesis has led to many interesting research questions that could not be fully investigated in this thesis due to their lack of immediate relevance, or their potential to open up larger research problems. A number of areas of potential future research have been identified for the K-Component model and CRL and are described below.

CRL Research Questions

In decentralised systems, the lack of a global view of the system generally prevents system designers from making prior assumptions about system properties. One alternative to making design time assumptions about system properties is to attempt to infer global properties from local observations. One potential approach would be to attempt to automate the tuning of parameters in CRL based on local observations. This approach is interesting to help cater for special cases, but in the general case there is the potential that errors in inferring global properties could propagate throughout the system leading to chaotic system behaviour. A possible way of ensuring that automated tuning of parameters meets the system objective(s) is to establish consensus on estimated local models of the system properties. Converged, estimated models offer a potentially interesting basis for designing algorithms to automate the tuning of CRL and reward model parameters to solve the optimisation problem at hand.

Also, decentralised systems may often require the optimisation of more than one system property. Multiple objective functions can be used to describe optimisation problems where there is more than one competing objective function, and further work is required on how to specify systems with many, possibly conflicting, optimisation goals in CRL.

In CRL, there is an assumption that every agent in the system is trusted. However, in open, distributed systems this will not always be the case. Future development of CRL could involve the development of local trust models of agents that have been encountered in the past. The trust models would, in effect, be a memory of previous interaction with those agents and trusted agents could also collaborate by sharing trust models of other agents. The reinforcement learning algorithm must also take into account the trust models when calculating the optimal policy and mechanisms could be developed to adjust the advertised function to take account of a neighbouring agent's trust profile.

K-Components

In K-Components, future work could involve migrating the framework to the web services platform where firewall traversal should not hamper the construction of large scale systems, as is currently the case with CORBA. There is scope for further research in simplifying the programming model. The K-Component model extends the CORBA programming model and provides a challenging programming environment. Adaptation contract programmers have to consider the interaction between adaptation contracts and component operation when defining action policies and designers of learning policies have to build a reward model, as well as model the dynamic behaviour of the component as a MDP. A simpler, declarative programming model could reduce development time for self-adaptive components.

The ACDL could also be integrated with more system support services for decentralised environments. For example, decentralised agents require service discovery capabilities to both bootstrap their operation and to update their view of available resources in their locality. The current version of K-Components does not provide support for decentralised service discovery and the addition of decentralised service discovery support, e.g., Universal Plug-and-Play for LANs or a service discovery engine for mobile ad hoc networks, would enable easier development of autonomic computing applications using K-Components.

Bibliography

A. Adi, A. Biger, D. Botzer, O. Etzion, and Z. Sommer. Context awareness in amit. In *Proceedngs of the Autonomic Computing Workshop, AMS '03*, 2003.

M. Agarwal, V. Bhat, Z. Li, H. Liu, B. Khargharia, V. Matossian, V. Putty, C. Schmidt, G. Zhang, S. Hariri, and M. Parashar. Automate: Enabling autonomic applications on the grid. In *Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services*, pages 48–59. IEEE Computer Society, 2003.

R. Allen. A formal approach to software architecture. In *Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144*, 1997.

R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE '98)*, 1998.

J. Almeida. Dynamic reconfiguration of object-middleware-based distributed systems. In *MSc. Thesis, University of Twente, Enschede, The Netherlands*, 2001.

A. Andrzejak, S. Graupner, V. Kotov, and H. Trinks. Adaptive control overlay for service management. In *Workshop on the Design of Self-Managing Systems, International Conference on Dependable Systems and Networks*, 2003.

ANTLR. Another tool for language recognition, version 2.4 for c++. In *http://www.antlr.org/*, 2003.

M. Appl and W. Brauer. Fuzzy model-based reinforcement learning. In *Advances in Computational Intelligence and Learning Methods and Applications*, The Kluwer International Series in Intelligent Technologies. Kluwer, 2002.

O. Ardaiz, P. Artigas, T. Eymann, F. Freitag, L. Navarro, and M. Reinicke. Self-organizing resource allocation for autonomic networks. In *1st International Workshop on Autonomic Computing Systems at 14th International Conference on Database and Expert Systems Applications*, pages 656–661, 2003.

M. Atighetchi. Building auto-adaptive distributed applications: The quo-apod experience. In *23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*, 2003.

A. Barabási. *Linked: The New Science of Networks*. Perseus Publishing, 2002.

A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. In *Discrete Event Systems Journal*, 2003.

A. Barto, R. Sutton, and C. Watkins. Learning and sequential decision making. In M. Gabriel and J. Moore, editors, *Learning and computational neuroscience : foundations of adaptive networks*, Cambridge, Mass, 1990. M.I.T. Press.

T. Batista, R. Cerqueira, and N. Rodriguez. Enabling reflection and reconfiguration in corba. In *The 2nd Workshop on Reflective and Adaptive Middleware*, 2003.

R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

K. Birman, R. van Renesse, J. Kaufman, and W. Vogels. Navigating in the storm: Using astrolabe for distributed self-configuration, monitoring and adaptation. In *Proceedngs of the Autonomic Computing Workshop, AMS '03*, 2003.

G. Blair, A. Andersen, L. Blair, G. Coulson, and D. Sanchez-Gancedo. Supporting dynamic qos management functions in a reflective middleware platform. In *IEE Proceedings - Software, Vol. 147, Issue 01*, February 2000.

G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of open orb v2. In *DSOnline*, volume 2, 2001.

G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, N. Parlavantzas, and K. Saikoski. A principled approach to supporting adaptation in distributed mobile environments. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 3. IEEE Computer Society, 2000. ISBN 0-7695-0634-8.

G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *Proceedings of the first workshop on Self-healing systems*, pages 9–14. ACM Press, 2002. ISBN 1581136099.

G. Blair, G. Coulson, and N. Parlavantzas. A resource adaptation framework for reflective middleware. In *Proceedings of the 2nd workshop on Reflective and Adaptive Middleware*, pages 163–168. PUC-Rio, 2003.

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

C. Boutilier, R. Das, J. Kephart, G. Tesauro, and W. Walsh. Cooperative negotiation in autonomic systems using incremental utility elicitation. In *Uncertainty in Artificial Intelligence*, 2003.

F. Brazier and J. Treur. Compositional modelling of reflective agents. In *Int. Journal on Human-Computer Studies*, volume 50, pages 407–431. Academic Press, Inc., 1999.

F. Brazier and J. Treur. Managing conflicts in reflective agents. In R. Dieng and H.-J. Mueller, editors, *Computational Conflicts: Conflict Modeling as a Primary Design Technique for Distributed Systems*, Lecture Notes in AI. Springer Verlag, 2000.

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

L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. In *IEEE Trans. Software Eng. 29(10): 929-945*, 2003.

G. Caro and M. Dorigo. Antnet: Distributed stigmergetic control for communications networks. In *Journal of Artificial Intelligence Research*, volume 9, pages 317–365, 1998.

W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Rule-based strategic reflection: Observing and modifying behavior at the architectural level. In *In Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, 1999.

W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Explicit architecture and architectural reflection. In *In Proceedings of the 2nd International Workshop on Engineering Distributed Objects*, 2000.

D. Chess, A. Segal, I. Whalley, and S. White. Unity: Experiences with a prototype autonomic computing system. In *International Conference on Autonomic Computing*, pages 140–147. IEEE Computer Society, 2004.

S. Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, Oct. 1995.

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

O. O. Concepts. *JThreads/C++, version 2.0.0b1*. Object Oriented Concepts, 2000. URL `http://www.orbacus.com/support/new_site/manual/4.2.0/jthreads/jtc.pdf,(AccessedAugust2004)`.

O. O. Concepts. *Orbacus for C++ and Java, version 4.2*. Object Oriented Concepts, 2001. URL `Feb2004,http://www.orbacus.com/support/new_site/manual/4.2.0/users_guide/index.html`.

R. Crites and A. Barto. Elevator group control using multiple reinforcement learning agents. In *Machine Learning, volume 33; 2-3*, pages 235–262. Kluwer Academic Publishers, 1998.

C. Cuesta, P. de la Fuente, M. Barrio-Solorzano, and M. Beato. Coordination in a reflective architecture description language. In *Proceedings of the 5th International Conference on Coordination Models and Languages*, pages 141–148. Springer-Verlag, 2002a. ISBN 3-540-43410-0.

C. Cuesta, P. de la Fuente, M. Barrio-Solorzano, and M. Beato. Introducing reflection in architecture description languages. In *IEEE/IFIP Conference on Software Architecture (WICSA3)*, volume 224 of *IFIP Conference Proceedings*. Kluwer, 2002b. ISBN 1-4020-7176-0.

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.

K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

E. Dashofy, A. van der Hoek, and R. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 21–26. ACM Press, 2002. ISBN 1581136099.

P.-C. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. In *International Conference on Distributed Applications and Interoperable Systems*, 2003.

T. De Wolf and T. Holvoet. Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control. In *Proceedings of IEEE International Conference on Industrial Informatics*, pages 470–479, 2003.

E. Dijkstra. Self-stabilizing systems in spite of distributed control. In *Commun. ACM*, volume 17, pages 643–644. ACM Press, 1974.

P. DiLascia. Performance optimization, controls versus components. In *MSDN Magazine*, June, 2004. URL `http://msdn.microsoft.com/msdnmag/issues/04/06/CQA/`.

M. Dorigo and G. D. Caro. The ant colony optimization meta-heuristic. In *New Ideas in Optimization*, 1999.

J. Dowling and V. Cahill. Dynamic software evolution and the k-component model. In *Workshop on Software Evolution, OOPSLA 2001*, 2001a.

J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *Proceedings of Reflection 2001*, 2001b. ISBN LNCS 2192.

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

J. Dowling, E. Curran, R. Cunningham, and V. Cahill. Using feedback in collaborative reinforcement learning to adapt and optimise decentralised distributed systems. In *IEEE Transactions on Systems, Man and Cybernetics (Part A), Special Issue on Engineering Self-Orangized Distributed Systems*, volume 35. IEEE, 2005.

J. Dowling, T. Schaefer, and V. Cahill. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In *Proceedings of Software Engineering and Reflection 2000, LNCS 1826*, 2000.

H. Duran-Limon and G. Blair. Reconfiguration of resources in middleware. In *The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, 2002.

K. Dutton, S. Thompson, and B. Barraclough. *The Art of Control Engineering*. Pearson, 1997.

C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. A platform supporting coordinated adaptation in mobile systems. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02)*, pages 128–137, Callicoon, New York, U.S., June 2002a. IEEE Computer Society.

C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. Utilising the event calculus for policy driven adaptation in mobile systems. In J. Lobo, B. J. Michael, and N. Duray, editors, *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, pages 13–24, Monterey, Ca., U.S., June 2002b. IEEE Computer Society.

C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD Record, 18(2):399–407*, 1989.

R. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures.* PhD thesis, University of California, Irvine, 2001.

T. Fitzpatrick. Design and application of toast: An adaptive distributed multimedia middleware platform. In *IDMS*, pages 111–113, 2001.

G. W. Flake. *The Computational Beauty of Nature.* MIT Press, Cambridge MA, 2000.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

A. Ganek and T. Corbi. The dawning of the autonomic computing era. In *IBM Systems Journal, 42-1, pgs 5–18*, 2003.

D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *ACM SIGSOFT Workshop on Self-Healing Systems*, 2002.

D. Gelernter and N. Carriero. Coordination languages and their significance. In *Commun. ACM*, volume 35, pages 97–107. ACM Press, 1992.

I. Georgiadis. *Self-Organising Distributed Component Software Architectures.* PhD thesis, Imperial College London, 2002.

I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM Press, 2002. ISBN 1581136099.

D. Goldin and K. Keil. Toward domain-independent formalization of indirect interaction. In *2nd Int'l workshop on Theory and Practice of Open Computational Systems (TAPOCS).* IEEE Computer Society Press, 2007.

C. Guestrin, M. Lagoudakis, and R. Parr. Coordinated reinforcement learning. In *Proceedings of The Nineteenth International Conference on Machine Learning*, pages 227 – 234, 2002.

P. Haraszti, T. Schäfer, and V. Cahill. The iguana experience: Meta-level programming in a compiled reflective language. In *Workshop on Experience with Reflective Systems.* Online Proceedings http://www.openjit.org/reflection2001/reflective-systems.html, 2001.

M. Hayden. *The Ensemble System.* PhD thesis, Cornell University: Dept. of Computer Science, 1997.

M. Henning and S. Vinoski. *Advanced CORBA Programming with C++.* Addison Wesley, 1999.

P. Hinnelund. Autonomic computing - a method for automated systems management. In *MSc. Thesis, Royal Institute of Technology, Sweden*, 2004.

J. Holland. *Hidden Order: How Adaptation Builds Complexity.* Perseus Publishing, 1996. ISBN ISBN: 0201407930.

IBM. The autonomic computing toolkit reference guide. In *www.ibm.com/autonomic*, 2004.

J. Jann, L. Browning, and R. Burugula. Dynamic reconfiguration: basic building blocks for autonomic computing on ibm pseries servers. In *IBM Systems Journal*, volume 42, pages 29–37, 2003.

M. Jelasity, A. Montresor, and O. Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems*, volume 2977/2004, pages 265–282. Springer, 2003.

L. P. Kaelbling, M. L. Littman, and A. P. Moore. Reinforcement learning: A survey. In *Journal of Artificial Intelligence Research*, volume 4, 1996.

T. Kalibera and P. Tuma. Distributed component system based on architecture description: The sofa experience. In *Proceedings of Distributed Objects and Applications.* Springer-Verlag, LNCS 2519, 2002.

J. Kennedy and R. C. Eberhart. *Swarm Intelligence.* Morgan Kaufmann, San Francisco, California, 2001.

J. O. Kephart and D. M. Chess. The vision of autonomic computing. In *IEEE Computer*, volume 36, pages 41–50. IEEE Computer Society Press, 2003.

K. Khare and R. Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 428–437. IEEE Computer Society, 2004. ISBN 0-7695-2163-0.

R. Khare. *Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems.* PhD thesis, University of California, Irvine, 2003.

G. Kiczales, J. Lamping, C. Lopes, C. Maeda, and A. Mendhekar. Open implementation guidelines. In *19th International Conference on Software Engineering (ICSE)*, 1997.

G. Kiczales, J. Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* MITPress, 1991.

T. Kielmann. Designing a Coordination Model for Open Systems. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061, pages 267–284, Cesena, Italy, 1996. Springer-Verlag, Berlin.

M. Killijian and J. Fabre. Implementing a reflective fault-tolerant corba system. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 154. IEEE Computer Society, 2000. ISBN 0-7695-0543-0.

F. Kon. *Automatic Configuration of Component-Based Distributed Systems.* PhD thesis, University of Illinois at Urbana-Champaign, 2000.

F. Kon, R. Campbell, and M. Roman. Design and implementation of runtime reflection in communication middleware: the dynamictao case. In *ICDCS'99 Workshop on Middleware*, 1999.

F. Kon, T. Yamane, C. Hess, R. Campbell, and M. Mickunas. Dynamic resource management and automatic configuration of distributed component systems. In *USENIX COOTS*, 2001.

J. Kramer and J. Magee. Analysing dynamic change in distributed software architectures. In *IEEE Proceedings on Software, 145(5), 146-154*, 1998.

R. Laddaga. Active software. In *1st International Workshop on Self-Adaptive Software (IWSAS2000)*, 2000.

R. Lavender and D. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Proc. Pattern Languages of Programs*, 1995.

B. Li. *Agiolos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

Z. Li and M. Parashar. Rudder: A rule-based multi-agent infrastructure for supporting autonomic grid applications. In *International Conference on Autonomic Computing*, pages 278–279. IEEE Computer Society, 2004.

M. Littman and J. Boyan. A distributed reinforcement learning scheme for network routing. Technical Report CS-93-165, Carnegie Mellon University School of Computer Science, 1993.

H. Liu and M. Parashar. Enabling autonomic, self-managing grid applications. In *Self-Star Workshop*, 2004.

H. Liu, M. Parashar, and S. Hariri. A component-based programming model for autonomic applications. In *International Conference on Autonomic Computing*, pages 10–17. IEEE Computer Society, 2004.

J. Loyall. Qos aspect languages and their runtime integration. In *Proceedings of LCR98, the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1998.

D. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *DIMACS Partial Order Methods Workshop IV*, 1996.

D. Luckham and J. Vera. An event-based architecture definition language. In *IEEE Transactions on Software Engineering, Vol 21, No 9, pp.717-734., September*, 1995.

P. Maes. Computational reflection. Technical Report 87.2, Vrije Universiteit Brussels, Artificial Intelligence Laboratory, Jan. 1987.

J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, 1995.

C. Mariano and E. Morales. A new distributed reinforcement learning algorithm for multiple objective optimization problems. In *Advances in Artificial Intelligence, International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI*, 2000.

N. Medvidovic, P. Oreizy, R. Taylor, R. Khare, and M. Guntersdorfer. An architecture-centered approach to software environment integration. In *Technical Report UCI-ICS-00-11, Department of Information and Computer Science, University of California, Irvine*, 2000.

N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, volume 26, pages 70–93. IEEE Press, 2000.

Microsoft. Com home page. In *http://www.microsoft.com/com/default.asp, Last updated: 01/08/2002*, 2002.

M. Mikic-Rakic and N. Medvidovic. Support for disconnected operation via architectural self-reconfiguration. In *International Conference on Autonomic Computing*, pages 114–121. IEEE Computer Society, 2004.

R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

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

K. Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, 1999.

A. Montresor, H. Meling, and O. Babaoglu. Load-balancing through a swarm of autonomous agents. In *Proceedings of 1st Workshop on Agent and Peer-to-Peer Systems*, 2002.

A. Montresor, H. Meling, and O. Babaoglu. Towards self-organizing, self-repairing and resilient distributed systems. In *Future Directions in Distributed Computing, A. Schiper et Al (Eds.)*, volume LNCS 2584. Springer-Verlag Berlin, 2003.

A. Moore and C. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, volume 13, pages 103–130, 1993.

R. Moreira, G. Blair, and E. Carrapatoso. A reflective component-based and architecture aware framework to manage architecture composition. In *Third International Symposium on Distributed Objects and Applications (DOA'01)*, 2001.

S. Neema, T. Bapty, J. Gray, and A. S. Gokhale. Generators for synthesis of qos adaptation in distributed real-time embedded systems. In *The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 236–251. Springer-Verlag, 2002. ISBN 3-540-44284-7.

T. Nowicki, M. Squillante, and C. Wu. Fundamentals of decentralized optimization in autonomic systems. In *Self-Star Workshop*, 2004.

OMG. The corba component model. In *orbos/99-07-01*, 1999.

OMG. The common object request broker: Architecture and specification. In *3.0.2 edition*, Dec. 2002.

P. Oreizy, M. Gorlick, R. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. In *IEEE Intelligent Systems*, volume 14, pages 54–62. IEEE Educational Activities Department, 1999.

O. Othman, C. O'Ryan, and D. Schmidt. The design of an adaptive corba load balancing service. In *IEEE Distributed Systems Online, vol. 2, Apr. 2001.*, 2001.

P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using qdl to specify qos aware distributed (quo) application configuration. In *Proceedings of ISORC 2000*, 2000.

T. Panagiotis, T. Demosthenis, and J.-K. Mackie-Mason. A market-based approach to optimal resource allocation in integrated-services connection-oriented networks. In *Operations Research*, volume 50, 4, July-August 2002.

M. D. Pendrith. Distributed reinforcement learning for a traffic engineering application. In *Proceedings of the fourth international conference on Autonomous agents*, pages 404–411. ACM Press, 2000. ISBN 1-58113-230-1.

L. Peshkin and V. Savova. Reinforcement learning for adaptive routing. In *In Proc. of the Intnl. Joint Conf. on Neural Networks, IJCNN*, 2002.

I. Prigogine and I. Stengers. *Order Out of Chaos.* Bantam, 1984.

T. A. Project. Xerces c++ parser, 2.4 release. In *http://xml.apache.org/xerces-c/*, Dec 2003.

A. Ranganathan and R. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *ACM/IFIP/USENIX International Middleware Conference*, 2003.

A. Rao. Load balancing in structured p2p systems. In *2nd International Workshop on Peer-to-Peer Systems*, 2003.

B. Redmond. *Supporting the Unanticipated Dynamic Adaptation of Object-Oriented Software.* PhD thesis, Trinity College Dublin, 2003.

C. Reynolds. Flocks, herds, and schools: A distributed behavioural model. In *ACM SIGGRAPH*, 1987.

M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. Technical report, University of Chicago, 2001.

M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. In *IEEE Internet Computing Journal*, volume vol. 6, no. 1, 2002.

M. Roman. *An Application Framework for Active Space Applications.* PhD thesis, University of Illinois at Urbana-Champaign, 2003. URL `http://choices.cs.uiuc.edu/gaia/papers/thesis-mroman.pdf`.

S. Sadjadi and P. McKinley. Transparent self-optimization in existing corba applications. In *International Conference on Autonomic Computing*, pages 88–95. IEEE Computer Society, 2004.

T. Schaefer. *Supporting Meta Types in a Compiled Reflective Programming Language*. PhD thesis, Trinity College Dublin, 2001.

A. Schaerf, S. Yoav, and M. Tennenholtz. Adaptive load balancing: a study in multi-agent learning. In *Journal of Artificial Intelligence Research*. Morgan Kaufmann, Los Altos, 1995.

J. Schneider, W. Wong, A. Moore, and M. Riedmiller. Distributed value functions. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 371–378. Morgan Kaufmann Publishers Inc., 1999. ISBN 1-55860-612-2.

M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. In *IEEE Trans. Softw. Eng.*, volume 21, pages 314–335. IEEE Press, 1995.

M. Shaw and D. Garlan. *Software Architecture: Perspecitves on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.

B. Smith. Reflection and semantics in lisp. In *Proceedings of the POPL*, pages 23–35. ACM, 1984.

P. Stone. TPOT-RL applied to network routing. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 935–942, 2000.

B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201889544.

J. Strunk and G. Ganger. A human organization analogy for self-* systems. In *Workshop on the Design of Self-Managing Systems*, 2003.

R. Sutton. Learning to predict by the methods of temporal differences. In *Machine Learning*, volume 3, pages 9–44, 1988.

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

C. Szyperski. *Component Software*. ISBN: 0-201-17888-5. Addison-Wesley, 1998. ISBN 0-201-17888-5.

R. Taylor and C. Tofts. Self managed systems - a control theory perspective. In *Self-Star Workshop*, 2004.

B. Technologies. Quo toolkit reference guide, release 3.0.11. In *BBN Technologies*, 2002.

B. Technologies. Quo toolkit users' guide, release 3.0.11. In *BBN Technologies*, 2002.

I. Technologies. *Orbacus Whitepaper*. IONA Technologies, 2003. URL `Feb2004,http://www.orbacus.com/support/new_site/pdf/OrbacusWP.pdf`.

P. Tuma and A. Buble. Open corba benchmarking. In *Proceedings of the 2001 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2001.

R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. In *ACM Trans. Comput. Syst.*, volume 21, pages 164–206. ACM Press, 2003.

W3C. Document object model (dom). In *http://www.w3.org/TR/PR-DOM-Level-1, June 1999, W3C Recommendation*, 1999.

M. Waldrop. *Complexity: The Emerging Science at the Edge of Order and Chaos*. Simon and Schuster, 1992.

C. J. Watkins and P. Dayan. Q-learning. In *Machine Learning 8:279-292*, 1992.

M. Wermelinger. *Specification of Software Architecture Reconfiguration*. PhD thesis, Universidade Nove de Lisboa, 2000.

K. Whisnant, Z. Kalbarczyk, and R. Iyer. A system model for dynamically reconfigurable software. In *IBM Systems Journal*, volume 42, pages 45–59, 2003.

S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart. An architectural approach to autonomic computing. In *International Conference on Autonomic Computing*, pages 2–9. IEEE Computer Society, 2004.

S. Wolfram. *A New Kind of Science*. Wolfram Press, 2002.

M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. In *Knowledge Engineering Review*, volume 10(2), pages 115–152, 1995.

J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. In *Theory and Practice of Object Systems*, volume 3. John Wiley, 1997.

Appendix A

Abbreviations

ACDL	Adaptation Contract Description Language
ADL	Architecture Definition Language
AMM	Architecture Meta-Model
CRL	Collaborative Reinforcement Learning
DOP	Discrete Optimisation Problem
DP	Decision Policy
ECA	Event-Condition-Action
K-IDL	Extended Interface Definition Language for K-Components
MDP	Markov Decision Process
MOP	Meta-Object Protocol
P2P	Peer-to-Peer
RL	Reinforcement Learning

Table A.1: Glossary of Key Abbreviations used in the Thesis

Appendix B

ArchReflect, ArchEvents and Configuration Interfaces

The ArchReflect, ArchEvents and configuration interfaces are defined in tables B.2, B.4 and B.6, respectively.

ArchReflect Interface
double bind_connector(const char* ConnectorID, const char* CompID) double bind_connector(int ConnectorID, const char* CompID)
double rebind_connector(const char* ConnectorID, const char* CompID) double rebind_connector(int ConnectorID, const char* CompID)
double replace_component(const char* CompID,const char* kref) double replace_component(int CompID,const char* kref)
KOM::ObjectID* get_component(const char* CompID) KOM::ObjectID* get_component(int CompID)
KOM::ObjectID* get_connector(const char* ConnectorID) KOM::ObjectID* get_connector(int ConnectorID)
double action(const char* ActionID, Priority e, CORBA::Short strategy)
void resolve_invoke_component_action(int op_id, const char* targetID, const char* CompID, Priority e, long jitter, CORBA::Short strategy, void* any)
double poll_state(const char* StateID)

Table B.2: The ArchReflect MOP operations.

ArchEvents Interface
double register_component(const char* CompID, KOM::Object*, bool is_remote)
double deregister_component(const char* CompID) double deregister_component(int CompID)
double register_connector(const char* ConnectorID, KOM::Object&)
double deregister_connector(const char* ConnectorID) double deregister_connector(int ConnectorID)
void register_event_local(const char* EvtID, const char* predicate, EvtHandler&)
void register_event_remote(CORBA::FeedbackMgr_ptr p, const char* ConnectorID, ListEvtRegistration&)
void deregister_event_local(const char* EvtID)
void deregister_event_remote(ListEvtRegistration& listEvts)
double bind_connector(const char* ConnectorID, const char* CompID) double bind_connector(int ConnectorID, const char* CompID)
double unbind_connector(const char* ConnectorID) double unbind_connector(int ConnectorID)

Table B.4: The ArchEvents interface.

Configuration Interface
double register_contract(KOM::Object* ContractID)
double deregister_contract(KOM::Object* ContractID)
void register_event_local(const char* if_name, const char* EvtID, const char* predicate, EvtHandler&)
void register_event_remote(CORBA::FeedbackMgr_ptr p, const char* ConnectorID, ListEvtRegistration&)
void deregister_event_local(const char* if_name, const char* EvtID)
void deregister_event_remote(ListEvtRegistration&)
void synchronise_feedback_states_with_meta_level()
CORBA::FeedbackMgr_ptr get_feedback_mgr()
char* get_comp_xml_desc(const char* CompID)
void register_advertisement(const char* int_s, const char* ad_xml, const char* ext_s);

Table B.6: The Configuration interface.

Appendix C

XML Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.dsg.cs.tcd.ie"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:dsg="http://www.dsg.cs.tcd.ie"
  elementFormDefault="qualified">
<!-- definition of root element -->
  <xs:element name="cb-values" type="dsg:conditions"/>
<!-- definition of attributes -->
  <xs:attribute name="value1" type="xs:double"/>
  <xs:attribute name="value2" type="xs:double"/>
<!-- definition of complex elements-->
  <xs:element name="unary-op" type="dsg:unary"/>
  <xs:element name="binary-op" type="dsg:binary"/>
<!-- definition of complex types -->
  <xs:complexType name="conditions">
    <xs:all>
      <xs:element ref="dsg:unary-op" minOccurs="0"/>
      <xs:element ref="dsg:binary-op" minOccurs="0"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="unary">
    <xs:attribute ref="dsg:op-name"/>
    <xs:attribute ref="dsg:value1"/>
  </xs:complexType>
  <xs:complexType name="binary">
    <xs:attribute ref="dsg:op-name"/>
    <xs:attribute ref="dsg:value1"/>
    <xs:attribute ref="dsg:value2"/>
  </xs:complexType>
</xs:schema>
```

Table C.1: Feedback Event XML Schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- KComponents-->
<xs:schema targetNamespace="http://www.dsg.cs.tcd.ie"
    xmlns:dsg="http://www.dsg.cs.tcd.ie"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<!-- definition of root element  -->
   <xs:element name="karchitecture" type="dsg:architecture"/>
<!-- definition of attributes -->
   <xs:attribute name="component-id" type="xs:integer"/>
   <xs:attribute name="component-name" type="xs:string"/>
   <xs:attribute name="connector-name" type="xs:string"/>
   <xs:attribute name="kref" type="xs:string"/>
   <xs:attribute name="poa-id" type="xs:string"/>
   <xs:attribute name="connector-id" type="xs:integer"/>
   <xs:attribute name="is-remote" type="xs:boolean"/>
   <xs:attribute name="cyclic" type="xs:boolean" default="false"/>
   <xs:attribute name="event" type="xs:string"/>
   <xs:attribute name="value" type="xs:double"/>
<!-- definition of complex elements -->
   <xs:element name="kcomponent" type="dsg:component"/>
   <xs:element name="kconnector" type="dsg:connector"/>
<!-- definition of complex types -->
   <xs:complexType name="event">
      <xs:attribute ref="dsg:event"/>
      <xs:attribute ref="dsg:value"/>
   </xs:complexType>
   <xs:complexType name="component">
    <xs:sequence>
     <xs:element name="provides" type="xs:string" minOccurs="1" maxOccurs="1"/>
     <xs:element name="state" type="dsg:event" minOccurs="0" maxOccurs="unbounded"/>
     <xs:element name="action" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
     <xs:element ref="dsg:kconnector" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
         <xs:attribute ref="dsg:component-id" use="required"/>
         <xs:attribute ref="dsg:component-name"/>
         <xs:attribute ref="dsg:kref" use="required"/>
         <xs:attribute ref="dsg:is-remote"/>
         <xs:attribute ref="dsg:poa-id"/>
  </xs:complexType>
  <xs:complexType name="connector">
     <xs:sequence>
        <xs:element ref="dsg:kcomponent" minOccurs="0" maxOccurs="1"/>
     </xs:sequence>
     <xs:attribute ref="dsg:connector-id" use="required"/>
     <xs:attribute ref="dsg:connector-name" use="required"/>
     <xs:attribute ref="dsg:cyclic" default="false"/>
  </xs:complexType>
  <xs:complexType name="architecture">
     <xs:sequence>
        <xs:element ref="dsg:kcomponent" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="dsg:kconnector" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
        <xs:attribute ref="dsg:component-id" use="required"/>
        <xs:attribute ref="dsg:kref" use="required"/>
  </xs:complexType>
</xs:schema>
```

Table C.2: AMM-DOM Configuration Graph XML Schema.