

# Self-Organisation in Agent-Based Mobile Computing

R.W. Collier, M.J. O’Grady, G.M.P O’Hare, C. Muldoon, D. Phelan, R. Strahan, Y. Tong  
*Department of Computer Science, University College Dublin, Ireland*  
*{rem.collier, michael.j.ogrady, gregory.ohare, conor.muldoon, donnacha.phelan,*  
*robin.Strahan, yanjun.tong}@ucd.ie*

## Abstract

*This paper describes two self-optimisation techniques that have been employed in the design of two exemplar agent-based mobile computing systems, namely Gullivers Genie and the Agents Channelling ContExt Sensitive Services (ACCESS) architecture. Specifically, we describe how agent cloning is used within Gullivers Genie as a basis for delivering a scalable architecture. Additionally, we describe the concept of collaborative agent tuning, a process in which multiple agents negotiate in order to optimise their overall performance while residing on a Personal Digital Assistant (PDA).*

## 1. Introduction

Autonomic Computing is an emerging research area that is concerned with the development of software systems that are self-optimising, self-configuring, self-healing, and self-protecting. Central to this vision is the notion of a “smart middleware” that permeates a software system providing key autonomic functions. One approach to the delivery of such middleware is through the use of Intelligent Agents.

In this paper, we discuss the autonomisation of agent-based applications via the implementation of two self-optimisation techniques: *agent cloning*, and collaborative *agent tuning*. We illustrate these techniques within the context of two mobile computing applications: Gulliver’s Genie [7] a mobile tourist guide, and the Agents Channelling ContExt Sensitive Services (ACCESS) architecture, an open agent-based architecture for the delivery of multiple location-aware and context sensitive mobile services [5].

## 2. Self-Optimisation via Agent Cloning

As the demand within large-scale agent-oriented applications increases, there is a vital need to ensure that individual agents do not become *overloaded*. Overloading occurs when a single agent is required to perform a number of tasks in excess of its own capacity.

At this point, two possibilities occur: either the agent continues as before, resulting in a significant degradation in the overall system performance; or the agent/system must engage in self-optimisation process, in which the system adapts itself to minimise the effects of the overloading.

Agent cloning [8] is the process by which an agent replicates itself. Self-replication and mutation are central to biological system evolution and consequently, autonomic software systems need to exhibit similar characteristics. Therefore, one approach to managing overloading is the use of agent cloning to replicate the overloaded agent. This results in increased processing capacity for this component of the system. In general, the conditions under which an agent should clone are:

1. The agent has become overloaded.
2. Adequate resources (e.g. memory, processing capacity) are available.

To illustrate this approach, we introduce Gulliver’s Genie, and describe how it utilises agent cloning to maintain optimal performance as the demand varies.

### 2.1. Introducing Gulliver’s Genie

Gulliver’s Genie [7] (henceforth referred to as the Genie) is a mobile application that focuses on the delivery of context-sensitive services to tourists as they explore some city or other outdoor environment. Two services have been implemented, namely navigation support and cultural content delivery.

The navigation service provides the tourist with an electronic map of their vicinity, which displays, amongst other things, the tourist’s current position. Delivering multimedia content about the attractions that the tourist encounters is the *raison d’être* of the Genie. All content is context-sensitive, that is, it is pertinent to the tourists’ current location and has been personalized in light of dynamic personal and cultural interest profiles maintained by the Genie.

## Architecture

In essence, the Genie conforms to a simple client/server architecture. The client component is installed on the tourist's PDA and connects to a back-end server via a wireless network. However, a more accurate description of the Genie is that of a Multi-Agent System (MAS) comprising a suite of strong intentional agents of the BDI family [3], all collaborating to deliver the required service to the tourist. These agents are spatially distributed with two being hosted on the tourist's PDA and the remainder on the server (Figure 1). The constituent agents are:

- *Spatial Agent*: Monitors the tourist's position and informs the other agents of any activity.
- *Cache Agent*: Given the limited memory resources of PDAs and the requirements of multimedia data, the effective management of the available memory is essential to the timely delivery of content.
- *Registration Agent*: Manages the agent community on the server and assigns agents to individual tourists.
- *Tourist Agent*: A Tourist Agent is assigned to tourists when they commence a session with the Genie. This agent, in coordination with the other agents, seeks to respond to current needs and to proactively anticipate future needs.
- *GIS Agent*: Based on a tourist's position, it constructs a model of their surroundings.
- *Profile Agent*: Maintains and dynamically updates user profiles as a result of any tourist interaction.
- *Presentation Agent*: Seeks to preempt future requests by caching presentations on the server.

## Implementation

Gulliver's Genie has been implemented on a HP IPAQ. GPS is used for position determination and a connection between the client and server is enabled via GPRS using HTTP. All agents have been implemented using Agent Factory [2][6].

## 2.2. Scalability Issues in Gulliver's Genie

Demand within the Genie is largely dependant upon the number of simultaneous users. Scalability tests have identified a number of bottlenecks, which arise as the number of tourists increase. In particular, the key bottleneck proved to be the Presentation Agent. This is due to the complexity and time-consuming nature of its task - namely the continuous preparation of potential personalized presentations that may be

required at some point in the future, as dictated by the tourists' activities. This activity is complicated by the dynamic nature of the user model, as the model is continuously updated and refined in light of the tourists' interaction with the Genie.



Figure 1. Architecture of Gulliver's Genie

## 2.3. Agent Cloning in the Genie

Upon becoming overloaded, an agent should check whether adequate resources are available, and if so, clone itself. Once cloned, the agent should assign a sufficient number of outstanding tasks to the clone, thus ensuring that it is no longer overloaded.

In the context of the Presentation Agent, overloading occurs where the number of pending presentation requests has increased beyond a given threshold. Once the agent perceives that this threshold has been breached, it adopts a belief that it is overloaded. This triggers a complex behaviour that is specified by the three commitment rules below:

```
BELIEF(overloaded) => COMMIT(Self, Now,
BELIEF(true), evaluateResources);
```

```
BELIEF(overloaded) & BELIEF(canClone) =>
COMMIT(Self, Now, BELIEF(true), SEQ(clone,
splitPendingJobQueue));
```

```
BELIEF(clone(?name)) &
BELIEF(splitJobs(?jobs)) => COMMIT(Self,
Now, BELIEF(true), inform(?name,
assignedJobs(?jobs))
```

Basically, if the agent is overloaded, it begins by evaluating the resources available (this is the first rule).

This results in the agent adopting one of two beliefs – either that it has sufficient resources to create a clone, **BELIEF**(canClone), or that it does not have sufficient resources, **BELIEF**(insufficientResources). If the former of these beliefs is adopted, then the second rule is triggered. This second rule triggers the cloning process, but only if the agent is still overloaded and sufficient resources are available. Specifically, this rule initiates a sequence of two activities:

1. it triggers the cloning process, and
2. upon completion of the cloning process, it reallocates the pending jobs between the agent and its clone.

The final rule completes the reallocation process by informing the clone of the pending jobs it has been assigned.

In this way, the Genie can autonomously and dynamically replicate agent components, thus alleviating some system bottlenecks. The cloned agent operates as a peer in the system. Thus, it advertises its availability to any relevant agents, allowing additional jobs to be allocated to it. Naturally, if the workload decreases, the need for cloned agents no longer exists, and the cloned agent may arrange its own termination. This is achieved via the following commitment rule:

```
BELIEF(cloneOf(?name)) & BELIEF(jobs(0))
=> COMMIT(Self, Now, BELIEF(true), SEQ(
inform(?name, terminating), terminate))
```

A number of advantages accrue from taking this approach. The Genie is inherently extensible. Enhancing and refining the agents' behavior is straightforward. More importantly, the issue of scalability can be easily and explicitly accounted for during all stages of the software development lifecycle. The Genie therefore addresses one of the core tenets of autonomic computing, namely that of self-configuration under varying and unpredictable conditions.

### 3. Self-Optimisation via Agent Tuning

Agents often cohabit with other processes/applications in resource-bound environments. Consequently, it is essential that agents, where possible, adapt themselves to maximise the performance of the overall system, whilst possibly degrading their individual performance.

One approach to achieving this local degradation is a technique that is known as *agent tuning*. The goal of agent tuning is the on-the-fly modification of an agent's run-time parameters in an effort to optimize the performance of the system for specific activities.

However, in many situations, this process cannot happen in isolation. For example, on a resource-bounded device, reducing the processing footprint of a particular application may require that all the agents residing on that device degrade their individual performance appropriately. This requires that these agents negotiate the modified run-time parameters that they will use to achieve the required overall system degradation.

To illustrate this approach, we introduce the ACCESS architecture and describe how it utilised agent tuning to optimise the performance of the ACCESS client, which is deployed on a PDA.

### 3.1. The ACCESS Architecture

The ACCESS architecture<sup>1</sup> is an agent-based architecture, which supports the development and deployment of context sensitive services using mobile devices such as PDAs. ACCESS has been realized as an extension of *Agent Factory (AF)*, a pre-existing framework that delivers structured support for the development and deployment of multi-agent systems [2][6]. ACCESS augments AF through the provision of a membrane of agents that facilitate the rapid prototyping of context-sensitive applications. These agents form a cohesive management layer into which multiple heterogeneous context-sensitive services may be plugged. This enables service developers to focus on the implementation of their service, rather than the infrastructure required to deliver it. Most significantly, ACCESS combines a set of agents that reside on a user's mobile device, known as the ACCESS Client, with additional agents that act as a gateway to any deployed ACCESS services [5].

### 3.2. Performance Issues in ACCESS

A key problem associated with the ACCESS architecture is the rendering of maps upon the PDA. Specifically, the speed at which maps are rendered has been shown to be dependant upon the *sleep time* [2] of the agents that are residing on the PDA.

Each agent is executed in a separate thread. The *sleep time* defines the length of time that the thread sleeps upon completion of a single iteration of the agent's control algorithm. A larger sleep time increases the amount of time that an agent's control thread is inactive. This leaves more time for other threads, such as the AWT thread (which monitors and updates the user interface), to be processed.

An analysis of how the sleep time affects the time taken to render a map has been carried out. Results of this analysis are illustrated below in figure 2.

---

<sup>1</sup> The ACCESS architecture received the System Innovation Award at Cooperative Information Agents (CIA2003).

This analysis shows that by varying the sleep time of one agent on the PDA, namely the Interface Agent (IA), it is possible to significantly improve the time taken to render a map. For example, increasing the sleep time of the IA from 1200 milliseconds to 1700 milliseconds caused the map rendering time to reduce from 60 seconds to 20 seconds. Additional increases caused this rendering time to be further reduced to approximately 12 seconds.

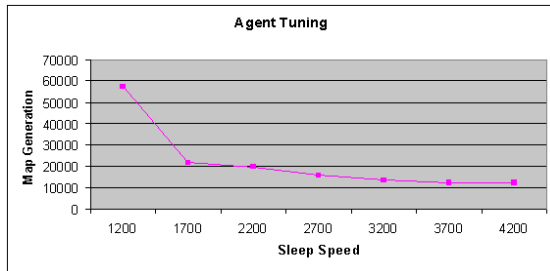


Figure 2. Map-Rendering Time Graph.

It is expected that other activities, such as the downloading of multimedia content into a cache or the migration of an agent to/from the PDA, will benefit similarly from the manipulation of the agent's sleep time. However, this issue is complicated by the expectation that a number of these activities will occur simultaneously, and further, may coincide with other activities, such as negotiation, whose performance improves as the sleep time decreases.

### 3.3. Collaborative Agent Tuning

Given the expectation that the ACCESS agents that reside on the PDA will perform a number of process-intensive tasks, it is vital that the agents be able to negotiate degradations, and also improvements, in one another's performance.

As shown in section 3.2, the sleep time is a key parameter that can be used to modify an agent's performance. However, any modifications to this parameter must remain within certain bounds. Specifically, the upper bound for sleep time (i.e. the slowest that an agent can operate) is based on a combination of:

1. the slowest speed at which the agent can react in a timely manner, and
2. the activities that the agent is currently engaged in.

Certain activities will require that the agent try to degrade the overall system performance, while others will require that the agent improve the overall system

performance. The result is that the agent must balance requests to tune themselves against their own requirements. Accordingly, we have developed a simple negotiation protocol that allows an agent to make a request that its co-habitants tune themselves, and which is informed of the outcome of each co-habitants decision. Being informed of the outcome of the tuning negotiation may result in additional rounds of negotiation, where the agent leading the negotiation tries to gain additional changes in performance from a subset of the agent community.

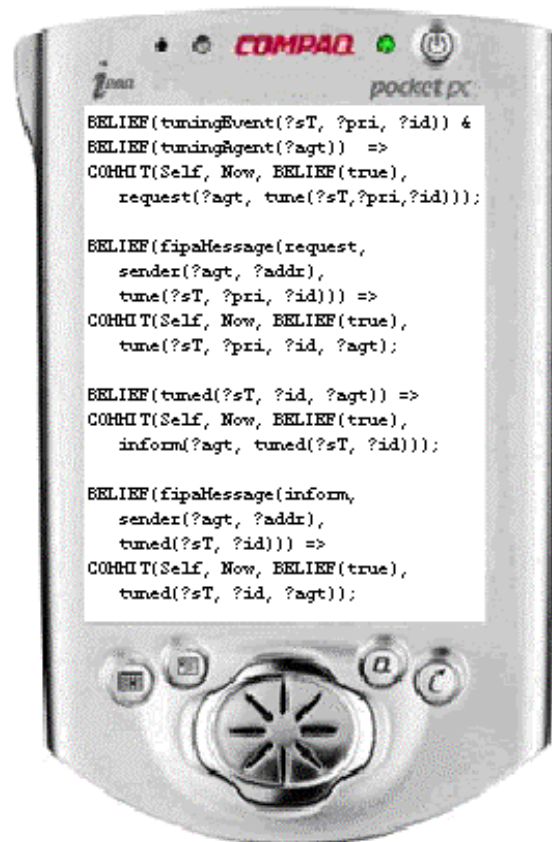


Figure 3. Tuning Agent Commitment Rules

Within ACCESS, support for collaborative agent tuning as been delivered through an agent role that specifies describes the protocol by which agents negotiate sleep time. This role has been implemented in AF-APL [2] and covers both initiating and responding to tuning requests. Informally, the role contains four rules. The first rule states that if the agent detects a tuning event with some sleep time, some priority, and some identifier, then it should request that all other tuning agents tune themselves according to the specified sleep time. The second rule describes how

the agent should respond to such a request. That is, it should commit to a “tune” action where by the agent uses a sleep time function to evaluate how to respond to the tuning request. Based upon the output of this function, the changes its sleep time (possible the sleep time requested), and generates a belief about how it has tuned itself. This causes the third rule to be fired, which informs the initiator of the tuning request of how the responder has modified its sleep time. The final rule deals with how the initiator deals with being informed of how the responder has reacted to the tuning request. Specifically the agent updates the tuning event, and if unhappy with the result of its request, triggers a second tuning event.

#### 4. Discussion

The cloning of agents has long been recognised as a valuable load-balancing technique [8]. However, empowering agents with system monitoring capabilities and the facility to self-replicate based upon the perceived system state, yields a powerful instrument for self-management.

Similarly, process prioritisation within distributed systems is not new. Agent based architectures already exist which, in part, facilitate user customisation of agent performance via a set of customisable parameters. Within ABLE [1] this is known as closed-loop control. Similarly the DIOS++ architecture [4] consists of sensors and actuators, which monitor and adjust system state. An autonomic agent permits self-management and dynamic adjustment of rules and policies at runtime to allow the system to alter and optimize its performance. These systems, while agent-based, differ from the approaches described in this paper, where autonomic functionality is integrated into the core system components.

#### 5. Conclusion

Mobile and Ubiquitous Computing places ever-increasing demands upon the underlying software environments. In particular, restrictions in memory and processor real estate demand that these systems are imbued with dynamic and opportunistic self-organising capabilities.

This paper examines two techniques that offer stepping-stones toward truly autonomic agent behaviour. In particular, agent cloning offers one approach to delivering self-organising and self-managing abilities. Conversely, agent tuning provides a mechanism by which agents can, through collaboration, degrade the performance of individual agents, while improving overall system performance.

The usage of these techniques has been illustrated within two agent-based mobile computing applications, namely Gulliver’s Genie and the ACCESS Architecture, both of which have been developed by the authors. While more evaluations are required, informal testing has indicated that these techniques offer a valuable first step in delivering autonomic constructs for mobile computing applications.

#### 6. Acknowledgements

Michael O’Grady gratefully acknowledges the support of the Irish Research Council for Science, Engineering & Technology (IRCSET) through the Embark Initiative postdoctoral fellowship programme. Gregory O’Hare gratefully acknowledges the support of Science Foundation Ireland under Grant No. 03/IN.3/1361 and Enterprise Ireland through grant ATRP/01/209.

#### 7. References

- [1] Bigus, J., Schlosnagle, D., Pilgrim, J., Mills, W., Diao, Y., ABLE: a toolkit for building multiagent autonomic systems, *IBM Systems Journal*, 2002.
- [2] Collier, R, O’Hare, G.M.P., Lowen, T, Rooney, C.F.B., Beyond Prototyping in the Factory of the Agents, *3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS’03), Prague, Czech Republic, 2003.*
- [3] Kinny, D Georgeff, M, Rao, A, A Methodology and Modelling Technique for Systems of BDI Agents. *In R. van Hoe, editor, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, 1996.*
- [4] Liu, H., and Parashar, M., DIOS++: A Framework for Rule-Based Autonomic Management of Distributed Scientific Applications, *Proc 9th Int’l Euro-Par Conference (Euro-Par 2003), LNCS 2790, Springer-Verlag, pp 66 – 73, 2003.*
- [5] Muldoon C., O’Hare G.M.P., Phelan D., Strahan R., Collier R., ACCESS: An Agent Architecture for Ubiquitous Service Delivery, *Proc 7th Int’l Workshop on Cooperative Information Agents (CIA2003), Helsinki, 2003.*
- [6] O’Hare, G.M.P., Agent Factory: An Environment for the Fabrication of Distributed Artificial Systems, *in O’Hare, G.M.P. and Jennings, N.R. (Eds.), Foundations of Distributed Artificial Intelligence, Wiley, NY, 1996.*
- [7] O’Grady, M.J., O’Hare, G.M.P., Just-in-Time Multimedia Distribution in a Mobile Computing Environment, *IEEE Multimedia, (to appear), 2004.*
- [8] Shehory, O., Sycara, K., Chalasani, P., Jha, S., Agent cloning: an approach to agent mobility and resource allocation. *IEEE Communications, 36 (7) .pp. 58-67, 1998.*