D.B.E.

Digital Business Ecosystem

Contract n° 507953

WP 23: Fitness Landscape Implementation

Del 23.1: DBE Fitness Landscape

Project funded by the European Community under the
"Information Society Technology" Programme.

Contract Number: 507953

Project Acronym: DBE

Title: Digital Business Ecosystem

Version: 00.01b

Deliverable N°: D23.1 (DBE Fitness Landscape)

Due date: 31/12/2004

Delivery Date: 7/1/2005

Short Description:

This document describes research and preliminary implementation work done on the implementation of the evolutionary architecture for the DBE. The main contributions of this deliverable include a design and architecture for capturing input data for the evolutionary architecture called the interceptor architecture, as well as the development of a decentralised optimisation technique, called collaborative reinforcement learning, for building self-organising distributed systems. The deliverable is composed of both this document and code.

Partners owning: TCD

Partners contributed: TCD

Made available to: The Project Consortium and the EU

Versioning

Version	*Date*	*Name, organization*	*Description*
1.0	7/1/2005	Jim Dowling, TCD	Implementation of Fitness Landscape

Quality check

1st Internal Reviewer: Miguel Vidal

2nd Internal Reviewer: Gerard Briscoe

3rd Internal Reviewer: Paolo Dini

Contents

Publications by TCD Associated with this Deliverable

From Bibliography: [1], [11], [41], [42], [43], [44]

1. Introduction

This document describes research and preliminary implementation work done on the Evolutionary Environment (EvE), formerly known as the Fitness Landscape, for the purpose of supporting distributed evolutionary computing in the DBE.

This document is divided between two separate research efforts in WP23, necessitated by the project timeline. Firstly, the document describes the data requirements for an evolutionary computing system in the context of the DBE architecture. Evolutionary computing models are based on data models, and an outline of the different possible data inputs here. This part of the document presents an implementation of a data extraction model for the DBE architecture, known as the interceptor architecture, and concludes with the preliminary design of the evolutionary architecture and a timeline for the implementation of the EvE for the second half of the DBE project.

Secondly, the document investigates the implementation requirements for a decentralised evolutionary environment by providing background research on self-organising distributed systems. Three different self-organising distributed systems are described within the context of a taxonomy used to describe self-organising distributed systems. From this research, a set of requirements for a decentralised self-organising distributed system are established. With these requirements, an implementation of a self-organising distributed system is described based on a technique called collaborative reinforcement learning. This technique is a candidate technology for self-organising infrastructural services in the DBE. An evaluation of the algorithm is presented in the context of one of the EvE use cases.

Related Work in the DBE to this Workpackage

The science group of DBE is working towards providing models for self-organisation and designing an evolutionary architecture to represent and solve different problems in the DBE and to build “intelligence” into services and tools in the DBE.

This workpackage, the EvE implementation workpackage, is concerned with implementing models for self-organisation and the evolutionary architecture designed by partners in SP4, with tasks such as the simulator of fitness landscape (WP9, C4) providing the design for the implementation of the EvE. The Memory and self-organisation task (WP6, S6) has contributed a preliminary design an EvE for the DBE, see [40] and Section 5.1. This document describes the preliminary implementation work for these models, but as an agreed design of the EvE has not been released by the time of this deliverable (month 14), implementation research on decentralised, self-organising systems and existing use cases was carried out in the course of this workpackage.

The work on a decentralised optimisation algorithm and background research on decentralised, self-organising distributed systems is applicable to two areas in the DBE – one existing proposed approaches to solving supply chain problems, and secondly to the decentralised peer-to-peer DBE architecture that will be designed in the second half of the project. The optimisation of workflows, see Section 2.1, has been identified as a Use Case in DBE Deliverable 18.1 by Dini in [52], and Surrey have investigated the supply-chain problems using global optimisation techniques, such as the Langevin Equation, in [55]. The work described in this document, however, is concerned with using decentralised optimisation techniques, where SMEs in the system take independent optimisation decisions.

2. Use Cases for the Evolutionary Environment in the DBE

This section describes three different use cases for adding intelligence and self-organisation to implementation artefacts in the DBE architecture. We restrict ourselves to three use cases although, while use cases for related intelligent services in the DBE can be found in Dini [52] and more information on DBE uses cases will be specified in FZI: B9-"Use cases definition" and by TCH and ITA: B29-"Functional analysis and specification".

2.1. Use Case 1: DBE Workflow Optimisation

In this section, we introduce the use-case of evolving the set of services in a workflow (or service chain), part of the Consume Service use case defined by Dini in deliverable 18.1 [52], and say that the evolutionary architecture should evolve the workflow to provide a fitter pool of services, we can pose a series of questions to help elucidate the issues involved:

- What does workflow evolution mean in the context of the DBE?
- Does workflow optimisation mean that an evolutionary architecture implementation (owned a specific SME provider) can take a definition of a workflow and evolve a set of services that implement the workflow? Clearly this is not possible for services defined in a structured programming language such as Java, although some research towards this goal is specified in [37].
- We see the problem of from an evolutionary computing viewpoint as an optimization problem, i.e., discover the set of optimal existing services in the DBE that would implement the workflow.

Taking, the view that the problem is modelled as an optimization problem, leads to a further question:

- What data is available describing available DBE services, users, etc that can be used by the evolutionary architecture to determine the optimal set of services in the workflow?

A schematic design of the DBE workflow use case is presented in Figure 1. It illustrates a service that is consumed by a client that is in fact a composed service consisting of a workflow of DBE services. An optimisation engine (the EvE) can be used to determine optimal services in the workflow using fitness input data and the set of services in the DBE. In the DBE there are two classifications of fitness input data that can be used by the evolutionary architecture:

1. Static Data, such as the SDL, BML from the DBE Language SP.
2. Dynamic Data, such as service/user feedback, fitness data, logged historical data.

This document describes in Section 4 how the acquisition of fitness input data for the evolutionary architecture is addressed by the interceptor architecture.

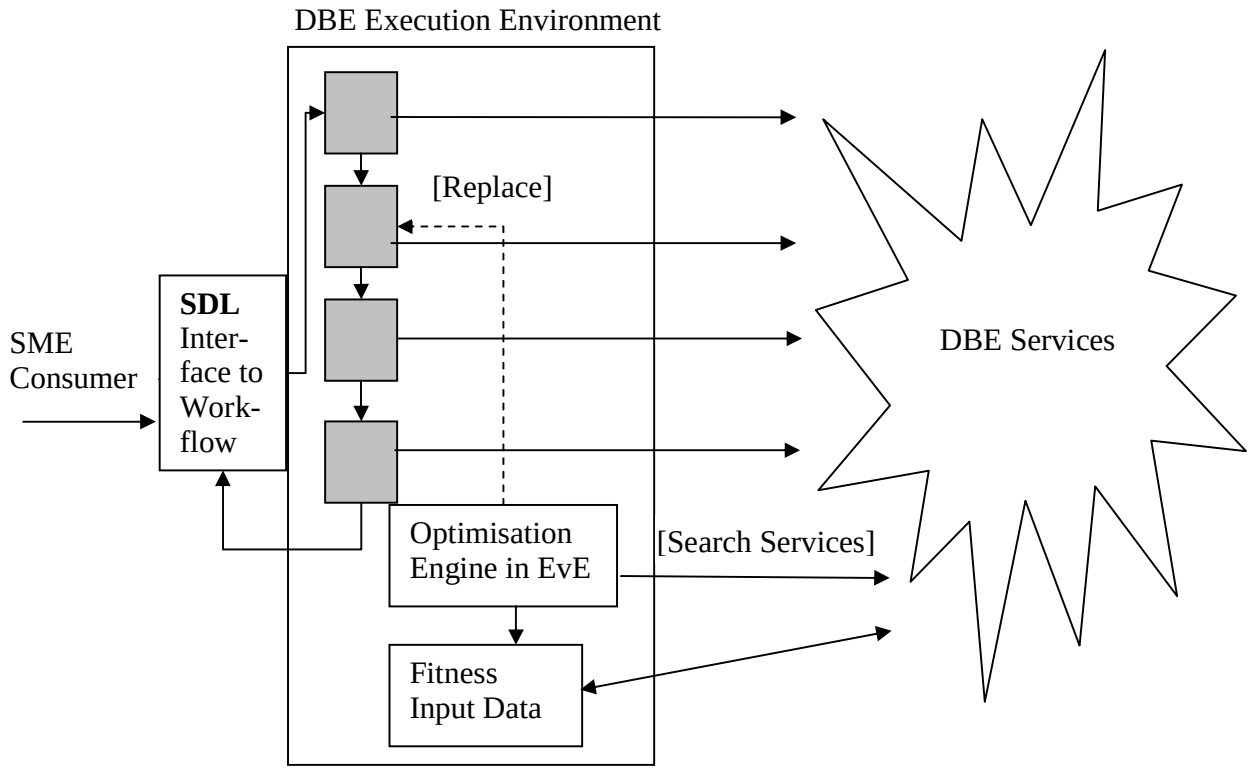


Figure 1 DBE WorkFlow Optimisation

2.2. Use Case 2: Optimising Stock Levels in a Supply Chain

SMEs that operate in supply chains and maintain stocks of goods have to address the problem of optimising the level of stock held in their inventory. In particular, they have to address the "bullwhip effect", see Section 8, that occurs when demand order variability in the supply chain is amplified as orders move up the supply chain, resulting in companies periodically hold excess and insufficient stock in their inventory.

The bullwhip effect is an emergent property of interacting companies in a supply chain. Previous attempts to reduce the bullwhip effect have concentrated on using information about inventory levels at companies higher-up in the supply chain to help determine optimal stock levels [59]. However, this approach is not possible for all SMEs, as they are often competing suppliers to the larger companies, that do not provide information about their internal stock levels.

This means that any attempt to solve the Bullwhip Effect problem for SMEs should rely on local information at SMEs and information other SMEs are willing to share with one another.

A decentralised optimization algorithm is also presented in Section 7, called Collaborative Reinforcement Learning (CRL), that is used to realise the stock level optimisation engine in this use case. An evaluation of CRL addresses the problem of optimizing the stock levels at SMEs is presented using a simulator that demonstrates the emergent property of a reduced Bullwhip Effect in supply chains for SMEs, see section 8.

2.3. Use Case 3: Optimising Routing to P2P Services in the DBE

The DBE distributed architecture will be built as a self-managing, self-organised peer to peer (P2P) system [37, 38]. The DBE architecture also contains many infrastructural, distributed services that have to be accessed by clients in the system [39], such as the Knowledge Base, Fada, the Distributed Storage Service and the Distributed Identity Service.

Use Case 2 describes the problem of how to optimise routing requests to infrastructural services in a P2P system without prior knowledge of service locations or the P2P network topology. The problem involves building a distributed system that allows the dynamic addition and removal of DBE infrastructural services from the P2P network, and using an optimisation engine at each peer, i.e. in the DBE Execution environment at each node, to build a self-organising routing protocol. This problem can be addressed using optimisation techniques such as evolutionary computing. In section 6.1 and section 7, we describe how we address the use case using a decentralised optimisation algorithm designed by TCD, called collaborative reinforcement learning. An outline design of how to address the problem is illustrated in Figure 2.

Other use cases that require optimisation and self-organisation at the DBE infrastructural level include optimising decisions on where to deploy an infrastructural service in the P2P system. Here traditional optimisation techniques such as dynamic programming are computationally intractable due to the dynamic structure of the P2P network, and without resorting to a fixed P2P topology, the problem could be addressed by framing it as how to solve the problem of optimally locating the services using fitness landscapes.

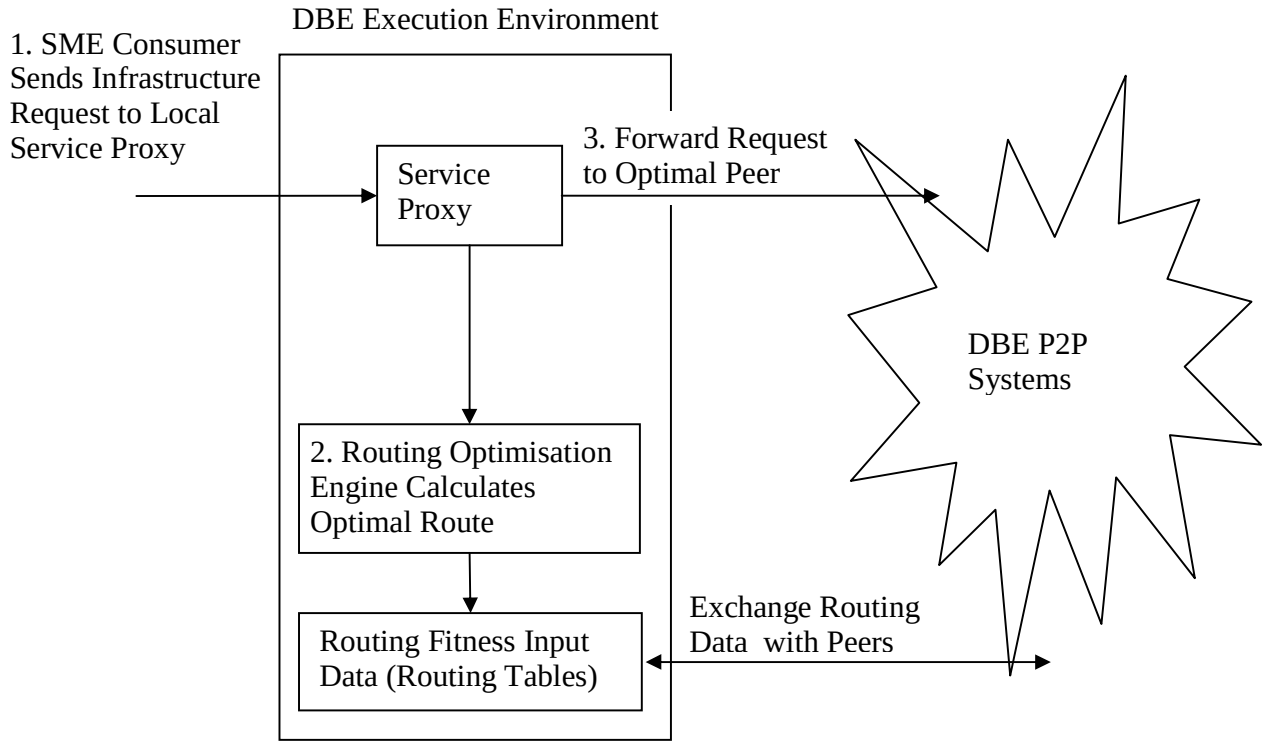


Figure 2 Optimising Infrastructure Service Routing Requests in DBE

This use case is related to the similar optimisation problem in a peer-to-peer network of where to optimally deploy instances of infrastructural services and where to store data in the system.

3. Evolutionary Computing

Evolutionary computing is part of computer science, not life sciences or biology [49], although it is inspired by biology and uses terminology similar to biology. A complete description of the evolutionary computing paradigm is not provided in this document, and the reader is referred to Eiben [49] for introductory material and Briscoe [40] for the design of an Evolutionary Computing Architecture (known as the Evolutionary Environment (EvE)) for the DBE.

3.1. Problem Solving using Evolutionary Computing

Evolutionary computation can be used to solve two main types of problems in the DBE [46]:

- Complex Optimisation Problems
- Machine Learning Problems

A classification of the types of machine learning or optimisation that can be performed by EC is adapted from [49] and illustrated in Figure 3:

- Type 1 systems describe systems where we have a model of the system and are looking for the inputs that give us a specified goal. Supervised Machine Learning, such as neural networks, are examples of Type 1 systems, and are not of great interest for building self-organising systems.
- Type 2 systems describe Evolutionary machine learning. We have corresponding sets of inputs & outputs and are looking for the model that delivers correct output for every known input. A possible application area in t the DBE is to build prediction models using historical data acquired using the Interceptor Architecture, see Section 4.
- Type 3 systems describe systems where we have a given model and wish to know the outputs that appear given different input conditions. They can be used to make predictions in dynamic environments
- Type 4 systems describe autonomous agent-based systems that take actions in an unknown environment and the environment supplies feedback about the “success” of the agent’s actions. They are not part of Eiben’s classification, but are described as learning agents that require critics telling the agent how it is doing [47]. Collaborative Reinforcement Learning is an example of a Type 4 system.

From Figure 4, we can see that Use Case 1 (Workflow Optimisation) can be modelled as a Type 4 problem (where the model is the optimal set of services that make up the workflow and it is unknown and the effect of replacing a service in the workflow is also unknown but can be computed from feedback).

From Figure 4, we can see that Use Case 2 (Stock Level Optimisation) can also be modelled as a Type 4 problem (where the model is the unknown optimal level of stock in the company’s inventory and the amount of stock to be ordered from a supplier after a customer order is also unknown).

From Figure 4, we can also see that Use Case 3 (Optimising Routing to P2P Services) can be modelled as a Type 4 problem (where the routing tables are the model and are unknown and outputs are received when packets are routed providing feedback).

Our proposed realisation of the use cases is based on using collaborative reinforcement learning (to solve Type 4 problems), however, this does not preclude other solutions based on the other three types of problems, such as evolutionary computing techniques.

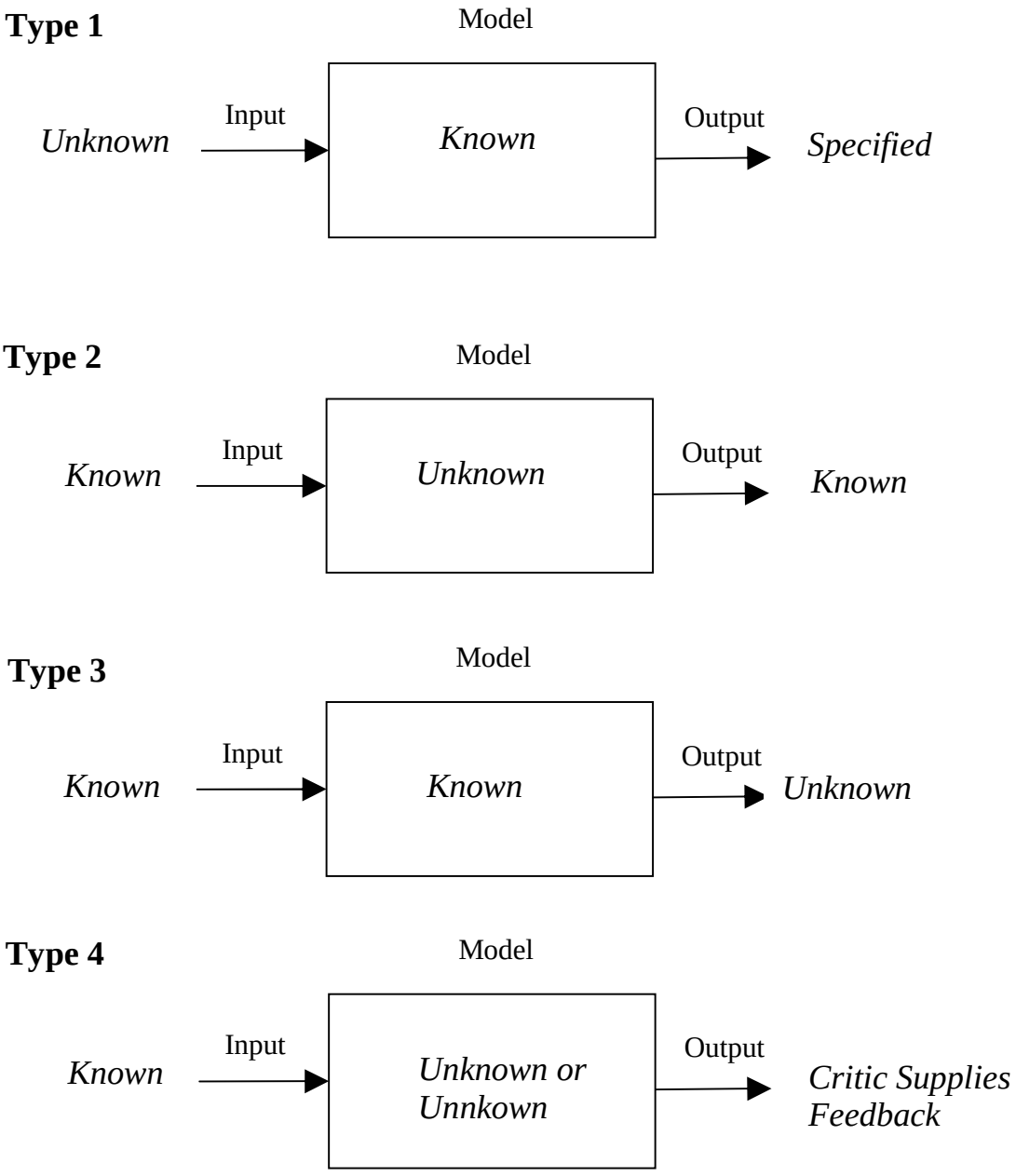
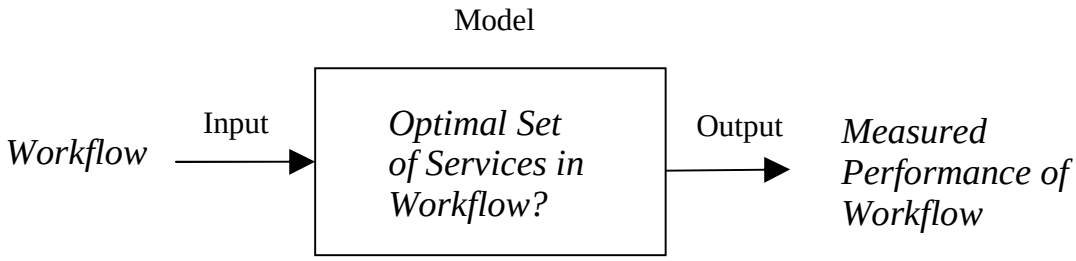
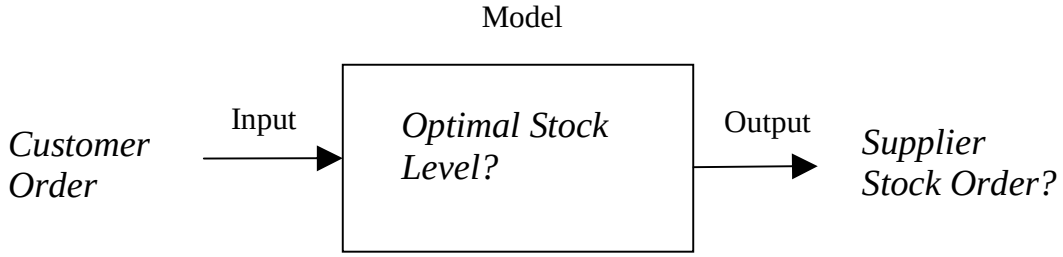


Figure 3 Categories of AI Problems addressed by Evolutionary Computing Systems [49]

Use Case 1: DBE Workflow Optimisation



Use Case 2: Stock Level Optimisation



Use Case 3: DBE Infrastructural Service Routing

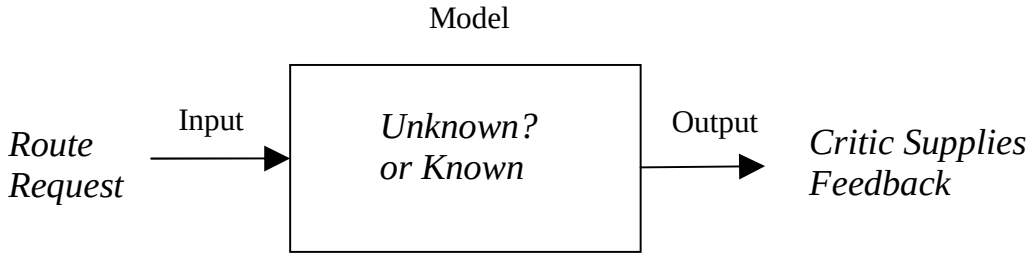


Figure 4 Categorising the Use Cases as AI Problems

3.2. The Evolutionary Environment Implementation in the DBE

A main challenge in designing the Evolutionary Environment (EvE) for the DBE is to set up a link between the EvE world where optimization can be performed and the DBE execution environment and system, where DBE services, providers and users reside. This is the problem of encoding data from the DBE world to the EvE, i.e., creating a representation of the problem so that EC techniques can be applied to solve the problem. This problem is addressed by the interceptor architecture in the next Section.

Secondly, the problem solving space where evolution will take place must be specified and designed. A preliminary design of the EvE has been provided by ICL and the implementation of the EvE will be performed in the second half of the DBE Project, see section 5.

Subsequent sections deal with the design and evaluation of a novel, decentralised optimised technique developed by TCD, called collaborative reinforcement learning, to solve use cases 1 and 2 in a decentralised P2P environment.

4. Interceptor Architecture

The interceptor architecture is designed to extract data from the DBE runtime for encoding to the EvE and other optimisation techniques that may be used in the DBE. The architecture is based on a generic interceptor mechanism in proxies that can be used to extract data flows through proxies and store selective data in persistent storage services in the DBE, such as the distributed storage service or local storage.

4.1. DBE Data Representation in the EvE

There are many different sources and types of data in the DBE, from service descriptions to service usage data, that can be used by the EvE and optimisation algorithms. For example, fitness landscapes of service and user reputation models require DBE data about historical transactions and service usage records in the DBE. There is a trade-off to be reached between the amount and sensitivity of the data that can be supplied by users and the DBE system and the power of the services and models we will be able to build.

In this section, we identify some of the data sources available for use inside the DBE by the EvE to solve previously identified use case problems and identify implementation problems associated with using the different potential data sources. Data can be anything from service usage statistics and feedback information on transactions in the DBE to derived data models, such as notions of trust, reliability even reputation. Data should also be classified as to whether it will be stored privately to DBE users, stored for public accessibility, not stored (i.e. shredded -) and whether the data will be stored anonymously or with user-/service-/company tagging.

Data Source

There are many potential sources of data in the DBE system. Static data such as BML (Business Modelling Language) descriptions, SDL (Service Definition Language) descriptions, Service Manifests (SM) is available from infrastructural services such as the Knowledge Base.

Dynamic and transient data, such as service requests, responses, faults, and composer queries can be extracted from the DBE messaging system using the interceptor architecture defined in this chapter, provided service providers and clients have authorised the extraction of the aforementioned data. Other application-specific data can also be derived from the more primitive data sources just mentioned.

Feedback data models, such as reputation information relating to companies, services and users, could also be a possible source of data for representation in the DBE. Feedback data models must balance the trade-off between proscribing actors in the DBE to supply mandatory feedback data and the willingness of the actor to accept the $cost_1$ involved in supplying that data. EvE models must ensure that the cost of acquiring the data is less than the reward DBE actors receive by using intelligent tools in the EvE built using the feedback data. As a didactic example, E-Bay requires mandatory feedback on the “success” of all transactions. This provides simple feedback about the reliability of buyers and sellers in E-Bay. A seller who has made hundreds of successful sales (a power-seller) is generally considered by E-Bay sellers to be more reliable than an unproven one. The power-seller can

charge a premium for goods based on his/her sales history. If, however, the feedback information were not mandatory would E-Bay users have willingly supplied it and would the reputation system have attained enough critical mass to be widely used?

4.2. Related Work on Interceptors

An Interceptor pattern allows services to be added transparently to a service or framework and triggered automatically when certain events occur. Interceptors represent an abstraction that solves the problem of tangled code dealing with security checks in the example below.

```
int send_data(Buffer *buf, size_t n) {
      log(buf, n);
      if(!check_security(buf, n)
            throw SecurityException(..);
      encrypt(buf, n);
      // etc...
      finally_actually_send_data(buf, n);
      return 1;
}
```

Here we have a network proxy function that has added code concerned with logging, security check and encryption. The code becomes tangled with the network proxy code and is not manageable or maintainable.

In the EJB Framework, this problem of tangled code concerned with security, transactions, logging, etc has been overcome using a general purpose interceptor architecture, see Figure 5.

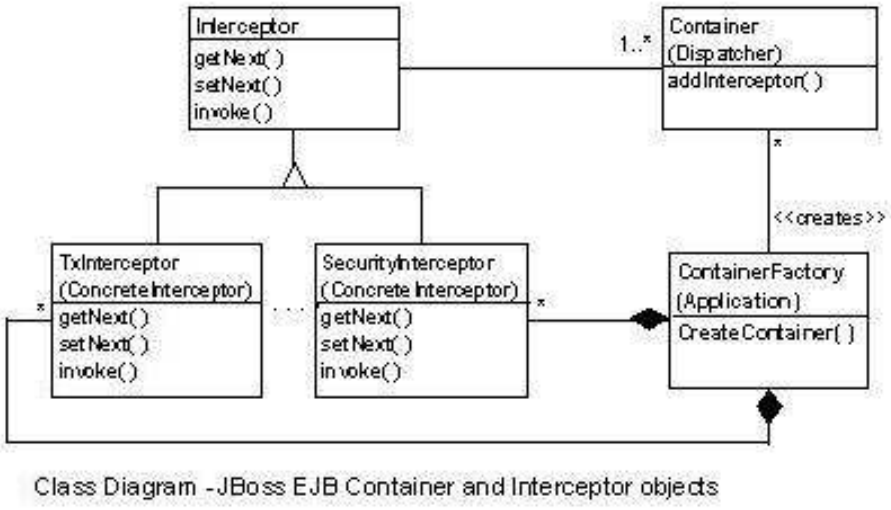


Figure 5 JBoss 2.3 Interceptor Model

When a ContainerFactory creates a Container, it registers a set of interceptors to each Container based on the order they are specified in the configuration file [UIUC-JBossInterceptor]. When a method is invoked on an EJB, the Container receives a MethodInvocation object and it invokes the first interceptor. It executes a single method on the object, invoke(), and passes it to the next interceptor by calling getNext(). All interceptors run on the order they were listed in the configuration file until the last interceptor is reached. Then, the Container is called, which invokes the business method originally requested on the bean. Examples of interceptor support in JBoss includes the LogInterceptor and SecurityInterceptor. In EJB interceptors are generated from an EJB's Deployment Descriptor.

Interceptors can be generated from different sources, but we expect that one main source of interceptor generation will be BML contractual agreements. If, for example, a user specifies in BML that the user of a service must agree to make information about the success (or not) of their usage of the service public, then we can generate interceptors from the BML description that will be part of the service invocation chain and can store information relating to service usage in distributed storage. The implementation of interceptors within the architecture may be developed using either dynamic proxies and before/after interceptor techniques. Before/after interceptors are implemented in at compile time, where the interceptor mechanisms are inserted into the byte code before a particular functional piece of code within a class and then again after this piece of code. This technique has been used in AspectJ and EJB architectures, while dynamic proxies are a feature of a Java that has been provided by the Dynamic Proxy API included since JDK 1.3. A dynamic proxy class is a class that can be used to create a type-safe proxy object for a list of interfaces without requiring pre-generation of the proxy class, such as with compile-time tools. Dynamic proxies can be used at runtime to insert interceptors between the method call and the actually execution of the method.

4.3. Interceptor Architecture for the DBE

The interceptor architecture is part of the DBE Architecture, as can be seen in Figure 6.

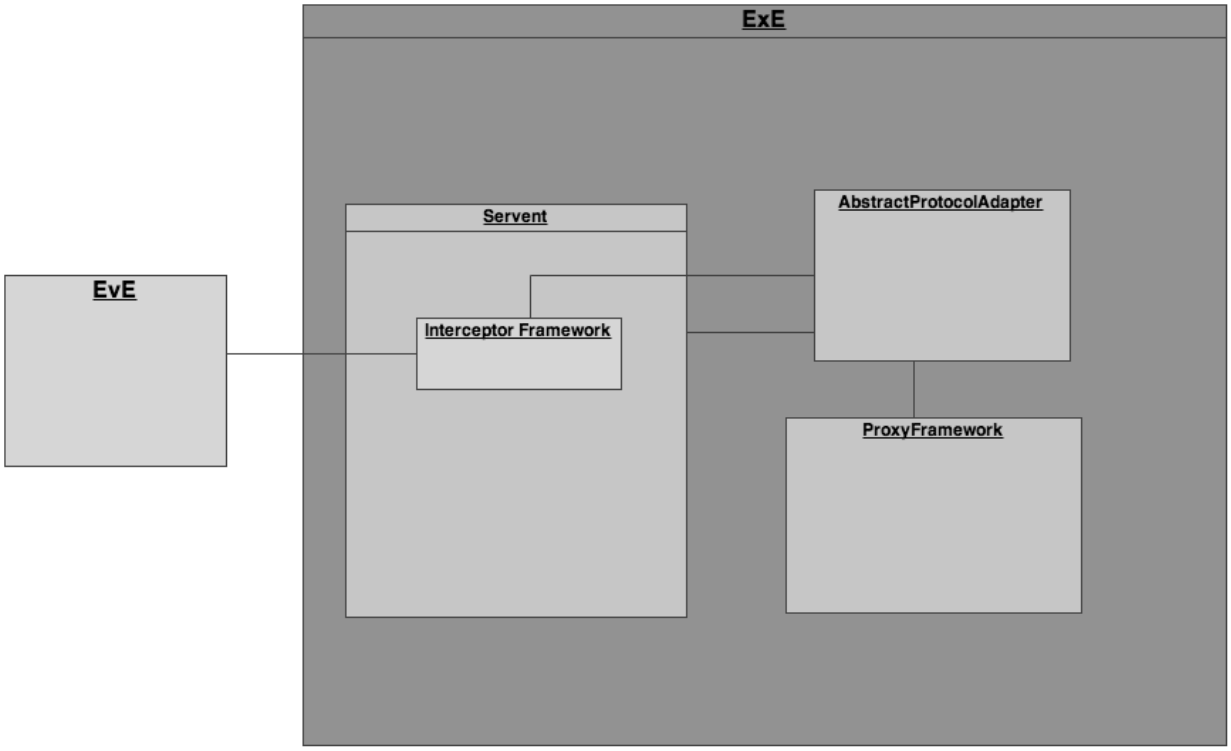


Figure 6 Interceptor Architecture in DBE Execution Environment [Adapted from Dini]

For the DBE servent, Apache tomcat was chosen as our application container [54] and for the first implementation of the DOP [39] messaging protocol, Apache axis SOAP messaging framework [54] was chosen. Within the axis framework, the messaging unit is a `MessageContext`. This `MessageContext` is the basic unit of work and it is upon these contexts, and the SOAP message within, that the axis engine processes message communications between local and remote processes. Within the axis engine, it is processing elements known as handlers that operate upon these `MessageContexts`. If we are to look and inspect the axis engine, it becomes quite apparent that the axis engine itself is a series of handlers arranged in a sequential chain. These handlers or interceptors allow the easy and custom extension of the axis messaging system at various different levels within the axis engine. To understand where interceptors fit into the axis architecture, the below diagram provides an overview of how a message is processed through the axis system on the server side.

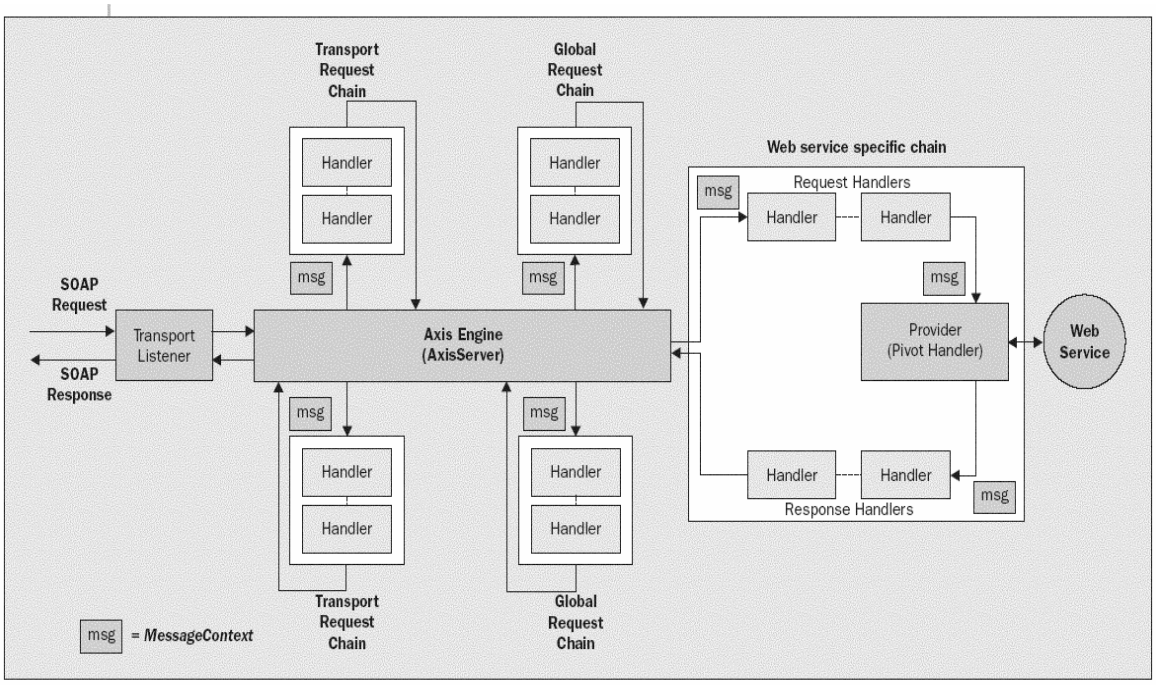


Figure 7 Handlers in the Axis Architecture

From the diagram in Figure 7, the transport listener wraps the SOAP request into a `Message` object, which is placed into a `MessageContext` object. Control is then handed over to the axis engine along with the `MessageContext` object. The next step is for axis to look up and determine if there is a transport request chain. If one exists it is invoked and the `MessageContext` object will be passed to it. This transport request chain will then invoke all handlers associated with it, passing the `MessageContext` along as a parameter, which the handlers will operate upon.

After the transport request chain has been invoked, the axis engine will then ascertain whether or not a global request chain has been configured to run. If there is one configured it will be invoked, and correspondingly the contained handlers, along with the `MessageContext` that was returned from the transport request chain.

Finally, the axis engine invokes the service request chain. This chain is a service-specific chain containing handlers specific to a particular web service deployed within the axis engine. In the web service request chain there is a special type of handler called the pivot hander. This is the handler that has the responsibility of making the actual call to the web service. In the axis engine there are two types of pivot handlers:

1 `RPCProvider` - This handler provides the functionality for the RPC-type communication semantics.

2 `MsgProvider` - This handler provides the functionality for the document-type communication semantics.

Once the appropriate pivot handler for the service has been invoked and a result from the

web service has been obtained, the axis engine invokes the chains in the reverse order to that of the request for the response back to the caller of the web service, that is the web service specific chain followed by the global response chain and then finally the transport response chain.

What the architecture described here gives us are multiple levels at which we can extract data and this data extraction can both be performed, with respect to the server side, on incoming messages (requests) and outgoing messages (responses). Broadly, we have three levels within the axis architecture to extract data at.

1. Transport level
2. Global level
3. Service level

We have mentioned the ideas of chains and handlers briefly. A handler is an object that interacts with a `MessageContext` and processes a message(s) contained within the `MessageContext`. A chain is also a type of handler, but it is a sequentially ordered collection of handlers where each handler is invoked in sequence. Figure 8 shows how handlers are related to one another.

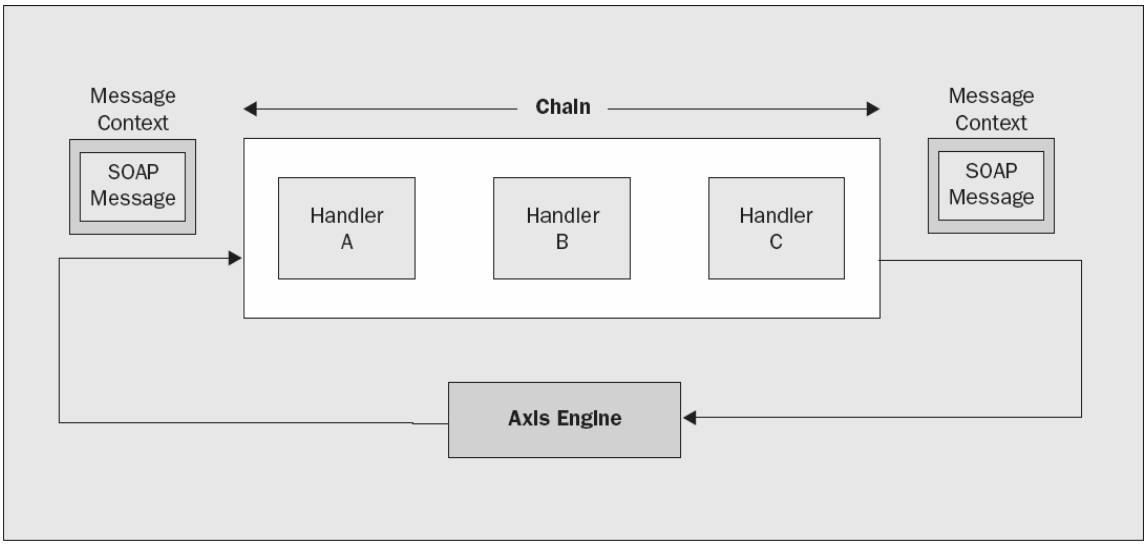


Figure 8 Handlers in the Interceptor Architecture

Figure 8 shows how the axis engine passes the `MessageContext` to the chain. The `MessageContext` contains the `Message`, which in turn contains the SOAP message. The chain will then call the invoke() method of each contained handler. In the above figure the processing order is A-B-C with each handler invocation returning control back to the calling chain. This sequence of method invocations can be shown by the sequence diagram in Figure 9.

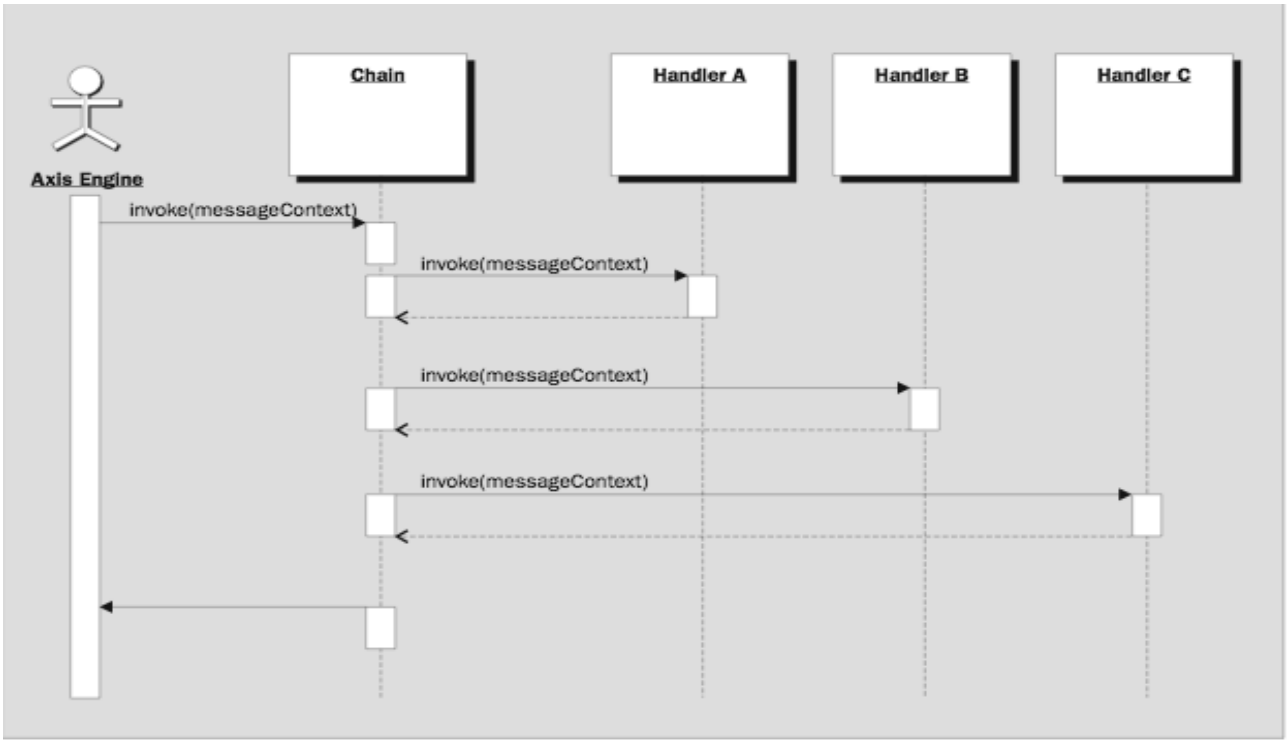


Figure 9 Chain of Handlers

4.4. Implementing Custom Handlers

In order to implement a handler that will fit into the axis handler architecture it is required that the implementing class inherit from the class org.apache.axis.Handler[1] and implement the following methods:

- `public void init()`
  This method allows the handler to initialise itself.

- `public void invoke(MessageContext mc) throws org.apache.axis.AxisFault`
  This is the main method where the handler performs its work. Here either the request or response message can be retrieve depending on whether the handler is in a request or response type chain.

- `public void onFault(MessageContext mc) throws org.apache.axis.AxisFault`
  If an error occurs within a chain the axis engine will invoke this method to

[1]It is also possible to inherit from the org.apache.axis.BasicHandler. This class provides basic implementations for life cycle methods init(), onFault(), cleanup(), leaving the implementation of invoke() up to the programmer.

undo any changes made by the handler.

- `public void cleanup()`
  This is called when the instance of the handler is no longer needed.

To configure the handler, the handler contains a `HashTable` of parameters as name-value pairs. This `HashTable` is populated by name and corresponding values contained in the web service deployment descriptor (WSDD) file, which is supplied to the axis engine on deploying a web service along with custom handlers.

4.5. Deploying Custom Handlers

To deploy a custom handler attached to a web service or indeed globally to the axis engine, the axis engine needs to be notified about the requirement. This is done by making an entry in the WSDD file. This entry is a XML element in the WSDD file and has the following syntax:

```
<handler name="handlername" type="handlertype">
      <parameter name="param1" value="value1"/>
      <parameter name="param2" value="value2"/>
</handler>
```

If there was the need for a chain of handlers then the above handler element would be wrapped in the following chain element so multiple handlers could be contained within the chain element.

```
<chain name="longchain">
      <handler>...</handler>
      <handler>...</handler>
</chain>
```

Now with the information on how to configure handlers and chains in the WSDD file, it is now necessary to describe how to attach the handler/chain as either a request or response handler/chain. This is done by wrapping either the handler or chain element in the following elements:

```
Request Chain/Handler:
      <requestFlow> ...chain/handler...</requestFlow>
Response Chain/Handler:
      <requestFlow> ...chain/handler...</requestFlow>
```

It is also possible to specify a fault flow, which tells axis how fault processing flow is to be directed. This is done using the `<faultFlow/>` element.

Finally, with a handler/chain configured as a response/request/fault flow it is necessary to specify at what level the handler/chain is to be inserted at. As described previously, handlers/chains can be inserted at three different levels; transport, global and web service level. These levels have the corresponding WSDD elements, which flow elements can be wrapped in:

```
<transport>
<globalConfiguration>
<service>
```

The configuration of both client and service side handlers follow the exact same WSDD syntax. The only difference in the two configurations is that the client configuration is read from a file (client-config.wsdd) that must reside on the class path of the client application.

4.6. Fitness Input Data Model and Interceptors

As previously mentioned, the interceptor architecture can be used to extract data from the DBE execution environment and store the data in persistent storage. As the interceptors process typed data, for the purpose of the Interceptor Demo, see Appendix A, a preliminary design of a custom type to describe fitness input data was developed, see Figure 10. The types supported include typical data types used by evolutionary computing algorithms, such as string, floats and ints.

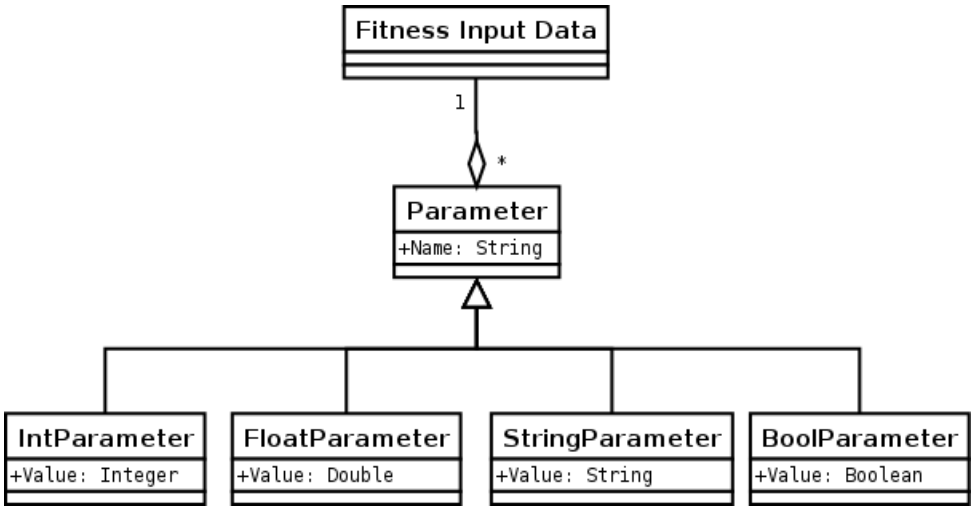


Figure 10 Fitness Input Data Model for Interceptor Architecture

5. Preliminary Design of the Evolutionary Environment

A preliminary design of an Distributed EvE for the DBE has been provided by Briscoe, Rowe and Dini in [40]. At present, the design is not at a level where work on the implementation can begin, as outstanding issues need to be reconciled between the model of the DBE envisaged by the Science group and its feasibility within the context of the architecture of the DBE specified by the computing group [39]. These issues are considered in the context of the timeline for the implementation of the Distributed EvE in Section 5.1. The timeline for the implementation of the Distributed EvE is presented in Table 1.

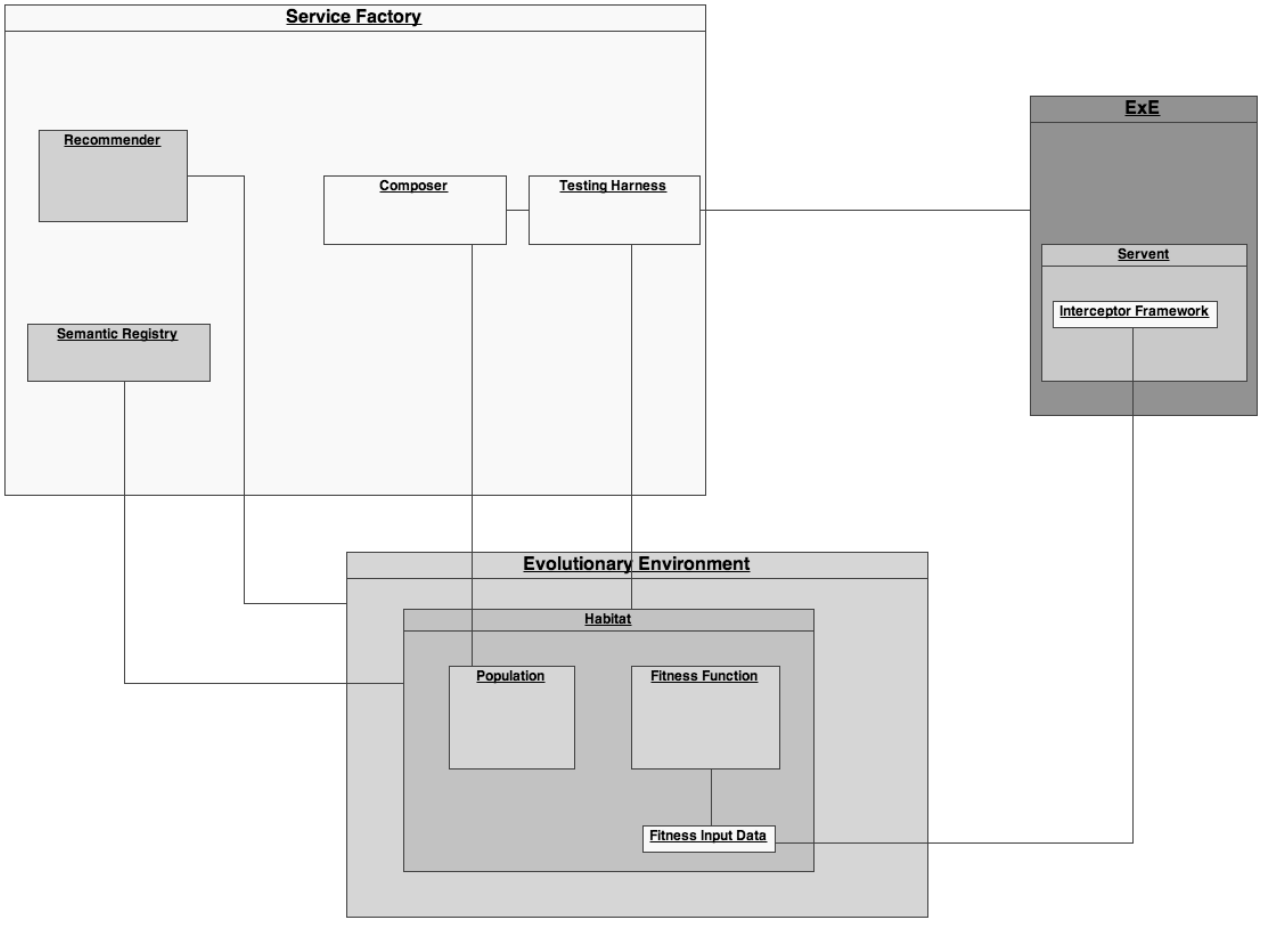


Figure 11 Evolutionary Environment in DBE Architecture

5.1. Timeline for Implementation of Evolutionary Architecture

Dates	Milestones, Achievements And Deliverables
January 2005 Month 14	Research on Implementation Issues for the Evolutionary Environment. Interceptor for architecture to link DBE Architecture and EvE. Collaborative Reinforcement Learning Model for Decentralised Optimisation. D23.1 Implementation of the Fitness Landscape
April 2005 Month 18	Evaluation of the Feasibility of the Design of the DBE Evolutionary Environment. Integration Plan for Evolutionary Architecture and DBE Execution Environment and Service Factory. M23.2 Design of the Distributed EvE
Nov 2005 Month 24	Evaluation of the Feasibility of the Design of the DBE Evolutionary Environment. Integration Plan for Evolutionary Architecture and DBE Execution Environment and Service Factory. D23.2 Design and Preliminary Implementation of the Distributed EvE
Feb 2006 Month 28	Functional Testing of EvE in DBE Architecture . M23.3 Functional Testing of Distributed EvE in DBE Architecture
April 06 Month 30	Implementation of the DBE Evolutionary Environment and Fitness Data Storage Service and Integration with the DBE platform. D23.3 Integration of the Distributed EvE to the DBE Architecture

Table 1 Timeline for Implementation of DBE Evolutionary Architecture

6. Self-Organising Distributed Systems

This chapter presents research on implementation issues for self-organising distributed systems. The issues covered here have direct relevance for the implementation of the EvE as a distributed Eve, as proposed Rowe in [56], as well as the implementation of the DBE as a self-organising P2P system.

Modern distributed systems have become more and more complex. Factors such as increased connectivity, in particular ubiquitous access to the Internet, diversity of devices, large numbers of users and available resources, high dynamism resulting from mobility of users and random failures have all contributed to increasing system complexity. However, traditional techniques for building distributed applications are proving inadequate in dealing with so high complexity and scale [21]. Hierarchical, top-down designed systems are often unable to adapt to extremely dynamic environment, since it is unfeasible to anticipate all possible states, interactions or failures of the system at the design time. The centralised approach, for example problem decomposition and modularization, is no longer appropriate for designing distributed applications as it requires too much a priori knowledge [16, 21, 41].

Most of existing systems, as they grow in size, require maintenance of skilled operators in order to function correctly. The emerging area of autonomic computing [19] envisions that soon systems will become too massive and complex to be manageable, beyond our ability to install, configure, optimise and maintain them. Consequently, system designers should tackle the problem of building self-managing, self-configuring, self-protecting and self-optimising systems.

A lot of recent research has focused on building such systems as decentralised, self-organizing systems [41, 16]. The advantages of such systems include extreme scalability, adaptability to changing environment, improved robustness, lack of critical points of failure or attack and self-management capabilities. It is our belief that such an architecture represents an appropriate model for both the distributed EvE and the DBE P2P Architecture.

Definitions of Self-Organising Distributed Systems

Research on the decentralised Systems has a long history, however, only recently self-organising mechanisms have been applied to these systems. The main difficulty in developing a decentralised system, sometimes composed of hundreds of thousands of entities, is the design of complex control mechanisms. According to [34]

"Decentralized control is using only local mechanisms to influence the global behavior. There is no central control, i.e. no single part of the system directs the macro-level [global] behaviour. The actions of the parts are controllable. The whole is not directly controllable".

Researchers observed that many decentralised mechanisms, which are self-organising, are prevalent in biological specifies in nature [7]. Examples include ant foraging behaviour, bird and fish flocking. These mechanisms are promising since they have evolved over millions of years and they have proved to be successful by enormous number of organisms. Their strength lies in the scalability, adaptability, flexibility and robustness. Distributed systems researchers have looked for the ways to apply these biology-inspired techniques to some difficult problems. Self-organisation seems to be especially suitable for decentralised systems where interactions between entities are often similar to those in the real-life organisms. Some of these mechanisms have already been applied to numerous problems with notable success.

For instance, mechanisms inspired by the ant colonies behaviour were used to build routing protocols for wireless [8] and telephone networks [29] that outperform existing routing protocols developed using traditional, modular design techniques.

Although research into self-organisation is inspired by observation and the study of natural systems, it has also gradually separated from it. Many self-organising decentralised systems are not anymore drawn from nature. Therefore, we feel the necessity to describe the common mechanisms useful when designing self-organising decentralised system. In our research, we were inspired by the definition of self-organisation proposed by Bonabeau [7]

"*Self-organisation is a process in which pattern at the global level of a system emerges from the numerous interactions among lower-level components of the system. Moreover, the rules specifying the interactions between the system's components are executed using only local information, without reference to the global pattern".*

A less specific, but useful definition was also proposed in [34]

"Self-organization is a dynamical and adaptive process where systems acquire and maintain structure themselves, without external control".

Since the latter definition does not expose the decentralised nature of these systems, which we think is essential, we prefer the former one. However, our studies on the state of the art self-organising systems induced us to propose another definition for self-organizing decentralised systems:

"Self-Organising decentralized system consists of interacting, autonomous components with only partial knowledge of the system that organise themselves without external control and which interactions provoke emerging consensus within groups of components on some properties of the system".

In the following subsections we describe the characteristics and mechanisms that are crucial when designing a self-organising decentralised system.

The following sections outline our taxonomy of self-organising distributed systems as a system consisting of:

1 Independent Agents

2 The Agent's Partial View of the System

3 Interaction Models for Agents

4 The Operating Environment of the Agents

5 Self-Organising Distributed Systems with Emergent, System Properties

5.1 Independent Agents

Every self-organising decentralised architecture is composed of a collection of, often simple, autonomous agents that collectively work to achieve a common goal.

Characteristics

A plethora of definitions and discussions emerged from the research on autonomous agents [14,17]. Based on research of existing self-organising distributed systems, see Chapter 6, we have identified the following characteristics of agents in a self-organising decentralised system:

- individual agents have incomplete capabilities to solve system problems

- each agent is autonomous in that it does not take any commands from some seen or unseen leader
- agents interact with each other and the environment in a proactive or reactive manner
- agents are capable of memorizing some of their previous interactions
- agents perform local computations that are asynchronous

Example Agents

Depending on the distributed application, an agent can take a variety of forms. An agent, for instance, can be a routing program residing on each agent involved in routing in a wireless ad hoc network as in the SAMPLE [11] system. Similarly, it can be a program participating in a gossip protocol as in Newscast [15] and Astrolabe [26] systems. An agent can also be a peer in a peer-to-peer systems such as FreeNet [9], CAN [25], Chord [31] or Pastry [28].

Competition vs. Cooperation

Both competition and cooperation between agents are mechanisms that have been used to self-organise decentralised systems [13]. Systems may support agents with conflicting interests, i.e., competitive agents that strive to maximise their individual performance, and self-organise through competition between agents [2]. Other systems support only cooperative agents and self-organise using cooperation between agents [1]. These systems prevent problems associated with greedy agents and can work well in environments that do not require human interactions that may introduce greedy behaviours.

In systems with competitive agents, Incentive mechanisms are used in order to cope with competitive agents. The incentive should be designed in such a manner that the best strategy for a selfish agent is to cooperate. This has been extensively studied adopting a game theoretic approach on a prisoners' dilemma problem [4,13]. Axelrod showed that the Tit-for-Tat strategy (i.e. an agent starts out by cooperating and then copies whatever action the opponent used last) dominates when the game has many iterations. This strategy was used in a popular BitTorrent system [10]. Other incentives for cooperation used in decentralised systems include reputation-based [6,5], payment-based [33] and market-based [32] approaches.

5.2 Partial System View of Agents

In decentralised systems no agent possesses a global view of the entire system as this is not feasible in any system with reasonable complexity [16, 41]. This is also the case of the DBE architecture, where no single DBE execution environment or EvE will have a global view of the system. Consequently in decentralised systems, agents have only partial (or local) views of the system. The agent's partial view of the system consists of the agent's neighbourhood, information perceived from the local operating environment, information obtained from neighbouring agents and estimations of some (unknowable) properties of the system.

Neighbourhood

The agent's neighbourhood is the set of agents it can directly communicates with at a particular moment in time, e.g. a peer’s adjacent peers in a P2P system. The agent may know

about more agents, but if it does not directly communicate with them, they do not belong to its direct neighbourhood. The neighbourhood set can be dynamic with respect to its size and the actual group of neighbours. It is delimited by the operating environment, the agent's capabilities and application specific properties.

In distributed systems an agent's network environment restricts the neighbourhood to reachable agents. For example, in a wide area network firewalls block the visibility of some agents. It often happens that two agents cannot directly communicate with each other because none of them have public IP address. In such situations these two agents cannot be direct neighbours without some bridging technology, e.g. web services. Similarly, in a wireless network the agent's neighbourhood is restricted to agents in a wireless communication range.

Since the agent is often not able to maintain and constantly update information about all potential neighbouring agents, it refines its neighbourhood to some limited number of agents. The neighbourhood can be refined using one of many existing proximity metrics such as the similarity of files' keys in the FreeNet [9] system, freshness of received updates in the Newscast [15] protocol, administrative domain membership in the Astrolabe [26] system or any other application-specific metric (e.g.. number of IP routing hops, geographic distance) as in the Pastry [28] peer-to-peer system. Other systems may not take proximity into account when refining the neighbourhood. For instance, agents in the Chord [31] structured peer-to-peer system select neighbours in a deterministic manner based solely on their unique identifiers.

In some environments, it is not necessary to refine the neighbourhood. For instance, the SAMPLE [11] wireless routing protocol utilizes the wireless medium to communicate with all agents in a wireless communication range and thus the neighbourhood consists of all reachable agents.

Some systems, on the other hand, do not have any neighbourhood at all. In the AntNet [8] mobile agent system there is no direct communication among agents. Instead, the communication is indirect and asynchronous, mediated by environment itself. This form of the communication is typical of social insects and is called *stigmergy*.

Representation of the World

Agents in decentralised systems build and constantly update their individual representations of the world. These representations are built through direct interactions with dynamic local environments and neighbouring agents. A local environment supplies application data such as routing packets in a routing protocol or sensor data in ubiquitous systems. Other information agents can acquire from the environment include reliability of network links connecting them with neighbours and availability of neighbours (i.e. their arrivals and retires from the system). Interactions with neighbouring agents serve a purpose of refining their individual models of the world.

The representation of the world maintained by an agent depends on the agent's application domain. In wireless routing agents, for instance, it can be composed of statistical models of quality of routes in a network [11], statistical models of route latencies [8], hop-counts to destinations [24,18] or battery levels at hosts [30]. The agent's model can also be composed of various estimated local or system properties [16,26]. In many peer-to-peer networks the system is represented as the routing table that enables peers to locate particular resources [9,31,28]. It is anticipated that in the next-generation ubiquitous computing the representation

of the system maintained by agents will be much more complex, composed of (among others) the location of the agent, time, activity of the agent, and accessible devices in its.

Necessity of Estimated System Views

It is essential to note that representations of the dynamic environment that agents maintain are only estimations since it is not feasible to update them as fast as new information arrives. Khare [20] noticed that it is not possible to guarantee simultaneous agreement for any centralised resource that changes more frequently than updates' dissemination time which is limited by the network latency. In decentralised systems, however, each agent is a potential source of new information and thus more suitable formula is:

$$d(x) > \frac{1}{f \times x} \Rightarrow \text{Synchronised Global View of System by Agents is not Possible}$$

where *f* is a rate of arrival of new information at one agent, *x* is a number of agents in the system and *d(x)* is a maximum time it takes to propagate a changed system view from any point in this system to all agents. It is worth noting that for any given system, when the number of agents grows the left-hand side of the formula increases (data propagation time increases), while the right-hand side of the formula decreases. Hence it can be always satisfied given enough agents in the system.

In a decentralised system, where this implication holds, agents often maintain outdated representations of the system. However, it is possible to enable a weak form of consensus to emerge in such systems. Numerous consistency models that have been proposed include probabilistic consistency which allows agents to make assumptions about other agents' partial views [16], eventual consistency which assumes that partial views eventually converge to one consistent state [26] or localised consistency model where consensus is achieved only within some limited group of agents [11].

5.3 Interaction Models for Agents

Interactions between agents are essential in any self-organising decentralised system. Agents need to continuously interact with each other in order to update their partial views of the system and consequently maintain consistent estimated system representations, adapt to the changing environment, optimise system's performance and finally produce a desired emergent global behaviour.

Updating Partial Views

The agent’s partial view is the agent's representation of its entire knowledge about a system. The agent can, however, only observe and experience some part of the system’s environment that he was assigned to. Therefore, in order to learn about other parts of the system, the agent has to communicate with other agents that have explored different parts of the system’s environment. Another possibility is that the agent is mobile in the system’s environment.

As systems with large numbers of agents produce frequent changes in the view of the system at different agents, agents continuously exchange with each other and update their local partial system views. Reasons for updating a partial view at an agent may include agents

neighbouring agents entering or leaving the system, and agent interaction with its environment to generate new local knowledge [7].

There are different existing techniques for exchanging partial views between agents in a decentralised system. Possible implementations include periodic broadcast or multicast as in gossip protocols [15,26], piggybacking advertisement in transmission or acknowledgement packets [11], event-based notification [31,28] and depositing advertisements in the environment through the stigmergy mechanism [8].

For the reasons in [20], agents are not able to update their partial views as frequently as the system's environment changes. Therefore, their partial views are only estimations that become invalid over a period of time and thus agents have to continuously maintain them by acquiring new information both from the neighbouring agents and local environment.

Figure 12 presents an example of decentralised multi-agent system. Agents in this system interact with their local environments and neighbouring agents in order to build and maintain their estimated partial views. Agent A has only two neighbours (agents B and C) it can share its estimated partial view with. Agents B and C can exchange partial views with each other, agent A and some other agents in the system.

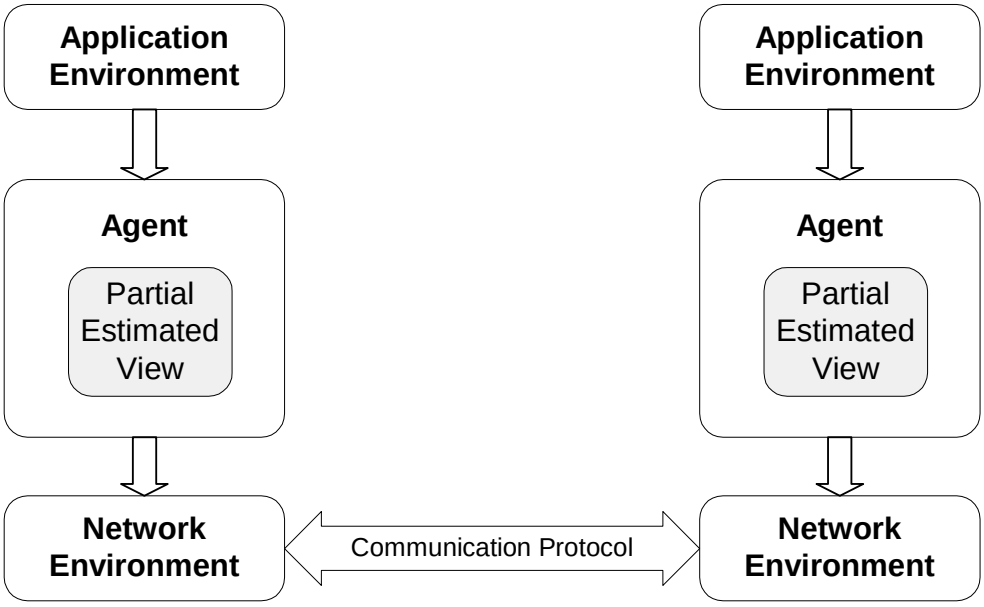


Figure 12 Agents interacting with local environment and neighbouring agents

Proactive vs. Reactive Interactions

Interactions between agents can be either proactive or reactive (i.e. on-demand). In a proactive protocol agents are interacting whenever they want in order to continuously update the accuracy of their partial views and self-organise themselves. On the other hand, in a reactive protocol agents do not autonomously trigger any interactions by themselves. The interactions are initiated only when some external source (such as the agent's environment) introduces information to the system. For example, in a reactive routing protocol a route for a packet is acquired only when an application requests that some packet is routed to a

destination [24]. In a proactive routing protocol the routes are actively maintained even if there is no traffic in the network [23].

5.4 Operating Environment for Agents

An agent in a distributed system always executes in a particular operating environment, that we call its environment. The environment determines the scope of agents' actions, their ability to sense and interact with the outside world (e.g., applications, the operating system and network). It also provides a flow of new data to the individual agents and to the system as a whole. The environment can be defined as the part of the system that is external to the agent and which has a direct impact on it.

Although the environment is domain-specific, it consists of similar elements in different decentralised systems. In most systems agents are controlled by users or applications build on top of them. From an agent's point of view these are part of its environment. For example routing agents are activated by both higher level applications and the network, which generate new packets for routing and also receive packets for forwarding or delivery. New packets can be seen as input data that is used to self-organise system. Similarly, in peer-to-peer systems users can share or search for resources by using lower level routing agents.

The environment also contains the system infrastructure with local resources such as file systems, databases and hardware devices. The network plays a special role among them, as it determines the agents' ability to communicate with other agents, in order to exchange and share information. The impact of the network architecture on the agents' interactions and the system properties is discussed more widely in the following sections.

Open versus Closed Environment

The environment in distributed systems can be open or closed. The former is characterised by agents' ability to join or leave the system, in the later the set of agents is fixed. A failure of an agent can be treated as a special case of agents leaving.

In self-organising systems reviewed in this document, the agent’s environment is open, the neighbourhood set of an agent can change any time. It is an agent's responsibility to maintain its neighbourhood view consistent with the current state of the neighbours.

Agents are usually discovered when they attempt to communicate. For example wireless devices can be detected by all other devices in the range when they emit a message. Routing nodes are detected when they send packets to the adjacent networks. In systems where broadcast is not possible, as in traditional peer-to-peer systems, agents who wish to be discovered may have to explicitly send notification messages to all potential neighbours or register in some, often hierarchical, name registry.

Agents leaving the system might use similar mechanisms - notify the neighbours just before it leaves or update the name registry. However, this mechanism is not sufficient in the case of failures, when agents can stop functioning unexpectedly at any moment. The only indication to other agents of such failure is absence of any activity of the failed agent, including communication. A common solution to this problem used in decentralised systems is called *heartbeat* technique. Agents emit a heartbeat message in every fixed interval of time, neighbours who do not receive the message after certain time assume the agent has died. A

dual approach depends on sending a *ping* message to the given agent, lack of response indicates a failure.

In SAMPLE [11] the agents use *decay*, all information received from neighbours degrades with time and is discarded after reaching some threshold value. This way agents who stop communicating become ``forgotten'' and removed from the system. A similar approach was applied in NewsCast [15].

Trusted versus Untrusted Environment

In some decentralised systems the environment is considered trusted, which means there is no need for any security mechanisms because all users who have access to the system are assumed to behave according to certain rules. However, it is more common in wide-area distributed systems to assume that the environment is untrusted. A popular approach, applied for example in Astrolabe [26], divides the system into *zones of authority*, where the environment (including other agents) within a zone is considered safe and it's untrusted outside of the zone.

Dealing with malicious agents is one of the most important security issues in untrusted environments. It should be distinguished though between two situations, when agents refuse to cooperate or act greedily, which is more a policy issue, and when agents maliciously try to breach the security using out-of-band mechanisms, violating the rules of the system, which is more close to traditional notion of security. Regardless of the behaviour of agents, an untrusted environment introduces security challenges such as secure communication, data confidentiality, authentication and authorization, non-repudiation, and others. In our research we discovered that most existing self-organising systems do not adequately address these security issues.

Network Architecture

The agent's communication ability largely depends on the network infrastructure. In this section we will briefly discuss the most common network architectures in existing decentralised systems.

Peer-to-peer and grid applications are usually deployed in wide-area networks (WAN). The basic properties of such networks are lack of broadcast facility and direct reachability of most nodes. The former has a strong impact on node discovery, two peers can only communicate if one of them possesses the address of the other one. The system's structure can grow when agents exchange their addresses but it is not clear how new agents can obtain a bootstrap (first) address in order to join the system. Another open issue is how to determine neighbours that are in close proximity.

Direct reachability of nodes in WANs implies that potentially any two agents can become neighbours. Firewalls often prevent this in practice and neighbour reachability may be only unidirectional, and if both nodes are hidden behind firewalls direct communication may not possible at all.

Mobile Ad-Hoc Networks (MANET), as well as sensor networks, are based on wireless communication media. The communication range of wireless devices is always limited to a certain distance, which forces the agents to perform cooperative multi-hop routing -- to

forward messages on behalf of each other. The communication range determines the neighbourhood of an agent.

Another important characteristic of wireless communication is the ability to broadcast and to receive messages from any device in the range. This allows agents to bootstrap into the system through easy agent discovery when agents are within communication range. The same functionality is available to agents in local-area networks (LANs). The communication range and the neighbourhood of a given agent is determined by the local networks the agent has access to. Broadcast operation in a local network is possible, hence also node discovery is feasible.

Bootstrapping

An interesting aspect is how agents in a self-organising system discover other nodes (bootstrap) when they join the system. It appears that it largely depends on the underlying network architecture. In a wireless network or a LAN (based on Ethernet) where broadcast is available, an agent can simply broadcast a join request and wait for a reply from any agent that heard it. In a wide area network it is not that trivial anymore. Many systems assume that an agent finds one or more other agents through out-of-band means. A system could, for example, use a name registry of well-known agents. This solution, however, conflicts with the concept of no shared global knowledge in decentralised systems. Despite that we will not discard these systems since bootstrapping depends on the low-level network properties.

5.5 Self-Organising Distributed Systems with Emergent, System Properties

A self-organising distributed system is a group of interacting agents functioning as a whole and displaying emergent system properties. There are many varieties of systems, on the one hand the interactions between agents may be fixed, at the other extreme the interactions may be unconstrained. Self-organising systems are those in the middle, with a combination both of changing interactions and of fixed ones. The system often has properties that are emergent, if they are not intrinsically found within any of the parts and exist only at a higher level of description. One of the example of emergent property is a *scale-free* network topology [3] that emerges from an initially random network, such as in Gnutella [27] or FreeNet [9].

Emergent Properties

Interactions between agents in a decentralised system are the mechanism that causes emergent properties to arise. An emergent property is a property that is not displayed in individual agents but is exhibited by a system as a whole.

The desired system properties emerge in many biological systems such as ant colonies that form shortest paths to food, birds' flocks and fishes in schools that coordinate their movements [7]. These emergent properties are not displayed in individual ants, birds or fish, as each “agent” only makes local decisions, but in the colonies, flocks or schools as a whole.

Emergent properties have been extensively studied by observing Cellular Automata (CA) [35] that are discrete dynamical systems whose behaviour is completely specified in terms of a local relation. CAs are interesting because despite being very simple to describe, they can

display very complex emergent patterns which cannot be deducted by examining rules controlling their individual behaviour.

In state of the art self-organising computer systems examples of emergent system properties include the emergence of shortest paths [8] or stable routes [11] to destinations in routing protocols for wireless ad hoc networks, the emergence of routing tables that allow fast lookup operation or an emergent network topology in a peer-to-peer information storage system [9].

Emergent properties, however, are often only estimations converging to optimal or near optimal results. The speed and accuracy of the convergence largely depends on the complexity of the problem and dynamism of the environment. In a highly dynamic environment, where the optimal result changes frequently, it is often not feasible to find it before it changes again. However, several systems allow us to make special presumptions about the estimated results. For example, SAMPLE [11] and AntNet [8] systems provide statistical guarantees, based on the history of interactions, that emerged stable and shortest paths respectively, are actually near to optimal. FreeNet's [9] lookup service provides only best-effort guarantees which mean that a request may not reach its destination.

Network Topology

Decentralised systems can have either structured or unstructured network topologies when connected topologies of agents are formed, in either a coordinated or uncoordinated manner. Structured decentralised systems include some peer-to-peer systems such as Chord [31] and Pastry [28]. Many self-organising P2P systems, such as Gnutella [27] and FreeNet [9], are initially unstructured (or random), but self-organise into scale free topologies [3] (i.e. its degree distribution follows a power law) that exhibit desirable *small-world* [22] characteristics such as high connectivity and small diameter. Scale-free topologies are an emergent system property that provides such systems with robustness to random nodes failures.

Evaluation Metrics for Emergent System Properties

The aim of self-organisation in decentralised systems is the emergence of system properties that optimise some system behaviour or provide some system property that provides some measurable benefit to the system. Since these systems have different applications, they aim to optimise system behaviours and may be evaluated using different performance metrics. For instance, an ad hoc routing protocol may strive to maximise system routing throughput [11], minimise routing latencies [8] or hop-counts [24,18], optimise power consumption in battery-powered nodes [30]. A peer-to-peer information storage system could aim minimise average resource lookup or insertion time [9].

6 Review of Existing Self-Organising Distributed Systems

Self-organising decentralised systems are used in a wide range of areas. They are applied to optimization problems [12], routing in ad hoc networks [11,1,8], CPU sharing, storage sharing and fast lookup services in popular peer-to-peer systems [9,25,31,28], aggregation in large-scale systems [16], distributed database, autonomic computing systems, data mining and management systems [26], and numerous other problems. In the following sections we review a selection of current state of the art self-organising decentralised systems, provide some characteristics and describe mechanisms that are crucial when designing such a system.

6.1 SAMPLE

SAMPLE [11] is a self-organising on-demand routing protocol for wireless ad hoc networks developed at Trinity College Dublin. It is novel in comparison to existing protocols in that it considers the quality of network links instead of solely minimising the number of routing hops to destinations, a metric favoured by existing protocols Ad Hoc On-Demand Vector Routing Protocol (AODV) and Dynamic Source Routing (DSR).

Agent

In the SAMPLE system agents are represented as a routing application running on each machine participating in a routing protocol. Each routing agent is autonomous since its decisions on where to forward a packet are made locally and are not directly influenced by other routing agents. Routing agents are assumed to cooperate and no incentive mechanisms are used. Therefore, if the protocol is deployed in an open environment, the protocol could be maliciously modified to produce routing agents that might prefer not to forward packets on behalf of other agents since it consumes their resources such as battery power. Unfortunately, none of the existing widely deployed wireless routing protocols cope with selfish agents.

Neighbourhood

The routing agent's neighbourhood in SAMPLE is delimited by the wireless communication range, as each transmission is promiscuously overheard by all agents in a communication range due to the shared wireless medium. The neighbourhood is highly dynamic since routing agents are often mobile in their environment and thus new agents constantly enter and exit every agent’s communication range.

Partial View

Each routing agent in the SAMPLE protocol has only a partial view of the network. The partial view consists of the agent's estimation of the quality of its links, based on an historical statistical model of traffic that attempted to use the link as well as information supplied by neighbouring routing agents about the quality of its links. Estimations of the quality of an agent’s links as well estimated route distances are used to determine optimal routes for packets to their destinations. Packets are routed using a probabilistic model to ensure a continuous level of exploration in the network to find new routes. An agent that is about to route for a packet, makes a decision of which link to route a packet probabilistically, choosing

more optimal links with higher probability. In order to restrict the size of the partial view and better adapt to the changing environment, a decay mechanism is used to gradually remove from partial views stale routes that are not used and thus not updated. Since each agent performs statistical estimations of links' quality individually, the partial views are not consistent among the set of agents the system. However, as neighbouring agents interact, exchanging partial views a localised consensus can emerge among groups of agents on stable routes in the system.

Interactions

The agent interacts both with the environment and neighbouring agents. The environment provides data packets to send and information about local network links quality (i.e. the number of retransmissions that were necessary to send a packet to a neighbouring agent). Since the agent learns directly from the environment only about the quality of local network links to neighbouring agents, it has to communicate with other agents in order to obtain estimates of system properties. Therefore, the neighbouring agents exchange parts of their partial views between each other. Each time an agent sends (or forwards) a packet, it piggybacks its estimated costs of delivering a packet to its origin and destination in terms of the overall number of retransmissions on these paths. Consequently, each neighbouring agent overhears these estimates and updates their own partial view by adding a statistical cost of forwarding a packet to this agent.

Environment

The SAMPLE routing protocol was designed to operate in an open and trusted Mobile Ad-Hoc Network (MANET). It is open since agents can join and leave the system seamlessly. Information about agents that left the system either on purpose or due to a failure is gradually removed from agents' partial views through the use of decay mechanism. Agents joining the system utilise the wireless broadcast mechanism to bootstrap their partial views with possible routes in a network.

System Properties

As routing agents in the SAMPLE system forward packets to popular destinations in the network, such as internet gateways, stable routes have been shown to emerge in the network [41]. This is achieved through positive feedback when agents discover more optimal routes and inform their neighbours, who in turn inform their neighbours until localised consensus on optimal routes has been reached. Negative feedback, such as when a route becomes overcrowded or link quality decreases, prevents routes, currently considered to be the most optimal, becoming overcrowded, by reducing their estimation of their quality. In such a case agents more willingly explore alternative routes to this destination.

The goal of the SAMPLE protocol is to use the emergent properties, such as stable routes, to maximise the overall network throughput, maximise the ratio of delivered packets to undelivered packets and minimise the number of transmissions required per packet sent. It has been shown to achieve this aim in [41], with significantly higher performance than existing protocols such as AODV and DSR.

6.2 Newscast

Newscast [15,16] is a scalable, self-organising distributed epidemic protocol for news dissemination. News are spread by allowing each process, engaged in the protocol, regularly contact arbitrary other process and execute the gossip protocol, during which the two exchange updates.

Agent

The agent in Newscast is represented as a process participating in the epidemic protocol. The agent is independent since it does not take any commands from any other agent. Agents are assumed to cooperate in news propagation.

Neighbourhood

The agent's neighbourhood is defined by the news it receives. Each news item is associated with a distinct agent that created it and a timestamp. The set of news' creators defines the neighbourhood. The least recent news item (i.e. with the oldest timestamp) is removed in order to limit the neighbourhood to a predefined constant size. The neighbourhood, however, is highly dynamic since agents regularly generate and exchange news with each other.

Partial View

The agent's partial view is composed of a random subset of all news items circulating in the system. It is build from the news received from neighbouring agents (but not necessarily produced by them) and news generated by local higher level applications. Two agents are guaranteed to have identical partial views only immediately after the gossip protocol, during which the two exchange news. However, as soon as any of these two agents execute the protocol again, its partial view will most likely be different again.

Each agent regularly generates updates and propagates them during the gossip protocol. However, the updates are not propagated immediately to all agents. Therefore, agents' partial views often contain outdated news. The eventual consistency model, used in the Newscast system, guarantees that in the absence of any further updates, all nodes should eventually reach the same state. In the case of Newscast, it means that if no agent produces updates, eventually all stale news in the system will be replaced by the most recent ones.

Interactions

Newscast agents interact with the local environment which is represented as some higher level application that generates and processes news. The agent is supplied with news that it is obliged to propagate in the system. The agent interacts with other agents in a proactive fashion, periodically initiating a gossip protocol with a randomly chosen neighbour. During the gossip protocol two agents exchange partial views and merge them with their current views by removing news with the least recent timestamp so that partial views have constant sizes. When the gossip is finished, the two agents should have equal partial views and neighbourhoods.

Environment

Newscast assumes that the environment is open and trusted. No security mechanisms against malicious or selfish agents are implemented. The membership protocol is simple and enables agents to join and leave the system at any time. To join a system, an agent contacts an arbitrarily chosen agent and copies its partial view. In order to leave the system an agent merely stops its communication. However, the method of discovering an initial agent is not specified. If the underlying network architecture does not provide an inexpensive broadcast or multicast mechanism (such as a wireless broadcast) then the centralised approach, such as a list of permanent agents, has to be used.

System Properties

The Newscast system maintains a constantly changing random communication graph over the participants. This graph has emergent properties that typify *small-world* [22] topologies, such as the small diameter (i.e. the maximum distance between any two nodes) and strong connectivity. Experiments proved that the protocol is able to sustain failures even up to 70% of all nodes. The small diameter, on the other hand, enables fast dissemination of news in the system.

6.3 FreeNet

FreeNet, as stated in [9], is ``an adaptive peer-to-peer network application that permits the publication, replication, and retrieval of data while protecting the anonymity of both authors and readers''. It can be seen as a distributed storage application which supports two operations: data insert and data retrieval. In addition, the system anonymity of both operations is guaranteed. Deletion of data is implicit; items which are not accessed for a long enough time are removed automatically. The system consists of a set of independent nodes, called also peers, running on separate machines under control of separate users. The peers communicate through the network using FreeNet's protocol. Each node contributes to the system by sharing part of its local storage space; the sum of the shares of all users determines the total storage space of the system. Each data item is identified by a key and is assigned to a peer; however, none of the nodes can obtain full information about the binding of keys to the peers. The nodes maintain local routing tables with limited number of entries and perform cooperative routing; a message can pass several nodes before reaching its final destination. Binding of keys to the nodes is dynamic; data is migrated and replicated transparently among the peers. Moreover, the replication is adaptive, replicas are created in those regions of the network where they are most frequently requested; unnecessary replicas are removed. There is no imposed structure of the network or routing information, the system is self-organising, classified also as an unstructured peer-to-peer system. By limiting information available to each node and by replicating data FreeNet provides anonymity for both publishers and readers. The peers can never discover the true origin of the data as well as the source and destination of transferred messages.

Agent

FreeNet, as all other peer-to-peer systems, by definition consists of peers, autonomous agents interacting through the network. The agents are functionally equal, there are no distinguished nodes (so called *super nodes*), or any other centralised facilities. The system

relies on the cooperation of agents -- without good will of the majority of users it cannot work correctly. However, the practice shows that the peer-to-peer applications work very well in the internet community, hence the assumption about the cooperation of users is satisfied.

Partial View

Each agent maintains its own local storage space, where other agents can insert files indexed by keys. An interesting property of FreeNet is that the nodes do not know the content of the data they store, it's encrypted. The only knowledge they have is the keys, which is necessary for data retrieval, this way the users are not liable for the content of the files. In addition to a local database, every agent maintains a routing table which enables locating files stored by other nodes. The table's size is limited, each entry contains a key and a node's address whom the agent forwards requests when looking for the corresponding key. If no entry matches a particular key, the agent uses the position with the nearest key in the table. This can be seen as estimation, when the agent lacks exact knowledge about a file it tries to approximate the location with the information it has. The accuracy of the tables delimits the quality of routing and the overall performance of the system, therefore the agents continuously try to improve it. In the forthcoming version routing protocol, it is planed to extend the information collected by agents with statistical data, including query response times, query success ratio and connection latency.

Neighbourhood

An agent's neighbourhood is determined by the content of its routing table, since the only nodes it can contact are those listed in the table. The neighbourhood is dynamic, new neighbours are added whenever new routes are established. At the same time when new neighbours are added some entries must be removed since the size of the routing table is limited. The entries are managed according to the LRU strategy (Least Recently Used), the oldest entries are deleted.

Interactions

The agents interact with users, who generate two kinds of events: file insert and file retrieval. These two operations are the only source of activity in the system, without them the system would remain frozen. In most cases the agents are not able to complete the user's request alone, thus they interact with each other. In the case of locating a file, the agent subsequently tries to forward the request to all it's neighbours until the file is found, starting from the nodes that are the closest to the requested key in the routing table. If none of the neighbours is able to find the file the operation fails. The maximal distance the query is propagated to is limited by *hops-to-live* parameter. The insert operation works in a similar manner, the file is forwarded between neighbours until it reaches a destination where it can be stored. A key issue of the algorithm is how the agents update their local views, i.e., how they update their routing tables and local databases. Whenever an agent returns a file to a neighbour, it provides also the original location where the file was obtained from. The recipient caches the file in its local storage and updates the routing table entry to point to the original location. This way the routing path to the file becomes shorter, and a new replica is created in a place where it was requested. It is important to note, that routing optimisation is performed reactively, when files are transferred. Without activity of the environment, such as requests

from the users, the system cannot optimise itself. The users are also the only source of new files, without the files the system cannot function.

Environment

The environment in FreeNet, as stated above, consists of the users and of the communication infrastructure, which is a Wide-Area Network (WAN), the Internet. The environment is open, agents can join or leave at any moment. Entering the system involves three phases. In the first the agent contacts any other node, which is currently implemented in a centralised manner, the agent receive a bootstrap file containing addresses of initial neighbours. It is an open problem how to provide decentralised node discovery (bootstrap) in a wide-area network. The second step is to initialise the routing table, which is solved by copying the table from the initial node. The third is to notify the network about the new node, which involves interactions between several agents and updating their routing tables. There is no dedicated algorithm for leaving the system. Agents that stop communicating are gradual removed from their neighbours' routing tables in result of the LRU strategy. In addition every agent discards neighbours that are not reachable when it attempts to contact them. The environment in FreeNet is untrusted. Any modification in a file’s contents, when the file is stored or when it's transferred through the network, can be detected by comparing its hash value with the key. A similar mechanism is used against overwriting. Decentralised replication prevents malicious removal of a file and denial of service (DOS) attacks. Similarly, a decentralised adaptive routing algorithm protects against malicious routing by single nodes.

System Properties

The goal of every agent is to find efficient routes to resources. The system's goal is to enable sharing of resources between agents, maximise system throughput, and provide anonymity at the same time. Routing is established in a self-organising way, since each agent acts independently without explicit knowledge of the system's structure. In fact there is no design of the structure, such as a ring with shortcuts in Chord [31] and Pastry [28] or coordinate system with zones in CAN [25]. The structure emerges from the interactions between agents in result of the self-optimising and self-repairing routing algorithm. Stable and fast nodes are promoted, as they manage to retrieve files more successfully. Agents tend to specialise in locating sets of similar keys. If a node is listed in routing tables under a particular key, it receives mostly requests close to this keys, and thus will become an ``expert'' in satisfying requests for those keys. Similarly, agents should specialise in storing files having similar keys, since they cache files they are requested for. Routing also adapts to the demand, keys that are more often requested in a particular area of the network are routed more accurately than other, less popular keys. This process can be described in terms of positive and negative feedback. The more a node becomes specialised in a range of keys, the more successful it is in locating files from this range, and the more often it is referenced in routing tables of other agents in positions corresponding to these keys. Deleting unused entries provides negative feedback. Overall throughput is maximised by adaptive replication, which self-organises in a similar way as routing. New replicas are created in areas where they are requested (positive feedback), unnecessary replicas are removed from areas where they are not used (negative feedback). Another emergent property of the system is scale-free (power-law connection distribution) network topology, which guarantees high robustness.

7 Collaborative Reinforcement Learning (CRL)

This section introduces collaborative reinforcement learning (CRL) as a technique for building self-organising distributed systems that can adapt and optimise system behaviours to a changing environment using positive and negative feedback mechanisms. CRL models the desired system properties as system optimisation problems that are solved by decentralised, collaborating routing agents that learn policies using reinforcement learning (RL) [9], [10]. RL is an unsupervised learning technique that allows an autonomous agent to monitor the state of its environment and take actions that affect its environment in order to learn an optimal policy. RL can be used to solve optimisation problems for an individual agent [9], but the application of RL to adaptively solve system optimisation problems in dynamic, decentralised environments is an open research problem.

CRL extends RL with different feedback models, including a negative feedback model that decays an agent's local view of its neighbourhood and a collaborative feedback model that allows agents to exchange the effectiveness of actions they have learned with one another. In a system of homogeneous agents that have common system optimisation goals and where agents concurrently search for more optimal actions in different states using RL, collaborative feedback enables agents to share more optimal policies, increasing the probability of neighbouring agents taking the same or related actions. This process can produce positive feedback in policy updating for a group of agents. The positive feedback mechanism reinforces changes in agent behaviour and can produce emergent consensus between agents on optimal policies. The positive feedback mechanism continues until negative feedback, produced either by system limitations or our decay model, causes agents to adapt their policies and often converge on stable, shared policies. Agents with converged policies can coordinate their actions to produce emergent, self-organised system behaviour.

7.1 Reinforcement Learning

In reinforcement learning, an autonomous 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 optimal policy using a reinforcement learning strategy [47,51]. The goal of reinforcement learning is to maximise the total reward (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. 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. Reinforcement-learning problems are usually modelled as Markov decision processes (MDPs). A MDP consists of a set of states, $S = \{s_1, s_2, \ldots, s_N\}$, a set of actions, $A = \{a_1, a_2, \ldots, a_M\}$, a reinforcement function $R : S \times A \to \Re$ and a state transition distribution function: $P : S \times A \to \pi(S)$, where $\pi(S)$ is the set of probability distributions over the set S.

7.2 Decomposing System Optimisation Problems

In CRL system optimisation problems are decomposed into a set of discrete optimisation problems (DOPs) [10] that are solved by collaborating RL agents. There are many system optimisation problems in distributed systems that can be naturally discretized into DOPs that

can be distributed amongst agents in a distributed system, such as load balancing of computation over a group of servers and locating replicated resources in a network. The solution to each DOP is initiated at some starting agent in the network and terminated at some (potentially remote) agent in the network. Each agent uses its own policy to decide probabilistically on which action to take to attempt to solve a DOP. In CRL the set of available actions that an agent can execute include *DOP action*s, A_{pi}, that try to solve the DOP locally, *delegation action*s, A_{di}, that delegate the solution of the DOP to a neighbour and a *discovery action* that any 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.

7.3 Heterogeneous Environments

In heterogeneous distributed systems, agents typically possess different capabilities for solving a given DOP. To model the differing capabilities of agents, CRL allows newly discovered agents to negotiate the establishment of *causally-connected states* with their neighbours by exchanging device capability information. Causally-connected states represent the contractual agreement between neighbouring agents to support the delegation of DOPs from one to the other. Causally-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 causally-connected to external states on many different neighbouring agents. An agent's set of causally connected neighbours represents its *partial-view* of the system.

In CRL, for every neighbour, n_j, with whom agent n_i shares a causally-connected state, there exists a delegation action, $a_j \in A_d$, that represents an attempt by n_i to delegate a DOP to n_j. If the delegation action is successful, n_i makes a state transition to its causally-connected state, s, terminating the MDP at n_i, and n_j initiates a new MDP to handle the DOP. 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 are modelled in the agent's reward model.

7.4 Model-Based Reinforcement Learning

RL strategies can be either model-free, such as Q-learning [49], or model-based [51]. Model-based learning methods build an internal model of the environment 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 [51]. 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 CRL algorithm.

7.5 Distributed Model-Based Reinforcement Learning

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_i, $V_j(s')$, and the connection cost, $D_i(s'|s,a) \in \Re$ where $a_j \in A_{di}$, to the next state when computing the estimated optimal state-action policy as $\mathrm{Q_i(s,a)}$ at agent n_i, see Equation 5. In the distributed model-based RL algorithm, our reward model consists of two parts. Firstly, a *MDP termination cost*, $R(s,a) \in \Re$, 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, $D_i(s'|s,a) \in \Re$, 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 to a neighbouring agent, while the connection cost for a transition to a local state after a delegation action should reflect 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 systems connections as rewards.

7.6 Advertisement

In CRL, neighbours are informed of changes to $V_j(s)$ of a causally-connected external state, s, at agent n_j using an *advertisement*. Each agent maintains a local view of its neighbours by storing *V* values for causally-connected states to neighbours in a local cache, $Cache_i$. The cache consists of a table of *Q*-values, for all delegation actions, A_d, at agent n_i, and the last advertised $V_j(s)$ for successful transition to the causally connected state. A $Cache_i$ entry is a pair $(Q_i(s,a_j), r_j)$, where r_j is the cached $V_j(s)$ value. When the agent n_i receives a $V_j(s)$ advertisement from neighbouring agent n_j for a causally-connected state s, it updates r_j in $(Q_i(s,a_j), r_j)$.

7.7 Dynamic Environments and Decay

Similar to RL, CRL models are based on MDP learning methods that require complete observability [51], 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 [51], and cached V_j values become stale using a *decay* model. In CRL, we decay cached V_j information in the absence of new advertisements of V_j 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 us to use 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 more dynamic network topologies.

7.8 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 [31], in the following schema:

- A dynamic set of *agents*, $N = \{n_1, n_2, \ldots, n_M\}$, often corresponding to entities in a distributed system.
- Each agent n_i has a dynamic set, V_i , of *neighbours* where $V_i \subset N$ and $n_i \notin V_i$.
- Each agent n_i has a fixed set of states S_i, where $S_i \in S$ and S is the system-wide set of states.
- Agents have both *internal* and *external states.*

  $Int : N \rightarrow \mathrm{P}(S)$ 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.
  $Ext : N \rightarrow \mathrm{P}(S)$ 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:

$$\begin{array}{l} Int(n_i) \subset S_i \\ Ext(n_i) \subset S_i \end{array} \quad \wedge \quad \begin{array}{l} Int(n_i) \cup Ext(n_i) = S_i \\ Int(n_i) \cap Ext(n_i) = \{\ \} \end{array} \tag{1}$$

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

$$\mathrm{Cn_i n_j} = \mathrm{Int(n_i)} \cap \mathrm{Ext(n_j)}, \text{ where } \mathrm{n_j} \in \mathrm{V_i} \tag{2}$$

  $\mathrm{s} \in \mathrm{Cn_i n_j}$ is a causally-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:

$$A_i = \{A_{di} \cup A_{pi} \cup discovery\} \tag{3}$$

- where $A_i \in A$, A_{di} are the set of delegation actions, A_{pi} are the set of DOP actions. The *discovery* action updates the set of neighbours, 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, A_{di} is updated to include a new delegation action that can result in a state transition to $\mathrm{s} \in \mathrm{Cn_i n_j}$ 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.
- $D: S_i \times A_{di} \times S_i \rightarrow \Re$ is the connection cost function that observes the cost for the attempted use of a connection in a distributed system. $D_i(s'|s,a) \in \Re$ 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 $Cache_i = \{\{(Q_i(s,a_j), r_j)\}: r_j \in \Re \wedge a_j \in A_{di}$. The r_j value 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 $n_j \in V_i$, $Cache_j$ is updated by a $V_j(s)$ *advertisement* for a causally-connected state. The update replaces the element r_j of the pair $(Q_i(s,a_j), r_j)$ in $Cache_i$ with the newly advertised $V_j(s)$ value .
- $\mathrm{Decay}(\mathrm{r_j}) \rightarrow \Re$ is the decay model that updates the r_j element in $Cache_i$,

$$\mathrm{Decay}(\mathrm{r_j}) = \mathrm{r_j} \times \rho^{td} \tag{4}$$

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, V_i, delegation actions, A_{di}, connected states, $\mathrm{Cn_i n_j}$ and its $Cache_i$. When a r_j entry in the cache drops below a specified threshold, the cleanup updater removes the delegation action a_d from A_{di}, the stale connected state s from $\mathrm{Cn_i n_j}$, and the pair $(Q_i(s,a_j), r_j)$ from $Cache_i$. If after removing s, $\mathrm{Cn_i n_j} = \{\}$ for some neighbour n_j of n_i, then n_j is removed from V_i.
- The distributed model-based reinforcement learning algorithm is:

$$Q_i(s,a) = R(s,a) + \sum_{s' \in S_i} P_i(s'|s,a) \cdot \left(D_i(s'|s,a) + \mathrm{Decay}(\mathrm{V_j(s')})\right) \tag{5}$$

where $a \in A_d$. If $a \notin A_d$, this defaults to the standard model-based reinforcement learning algorithm with no connection costs or decay function [51]. $R(s,a)$ is the MDP termination cost, $P_i(s'|s,a)$ 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 $a \in A_d$, and $V_j(s')$ 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 a neighbouring agent.

- The value function at agent n_i, $V(s)$, can be calculated using the Bellman optimality equation [48]:

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

7.9 Adaptation and Feedback in Distributed Model-Based Reinforcement Learning

Adaptation of system behaviour in CRL is a feedback process in which a change in the optimal policy of any RL agent, or a change in the system's environment as well as the passing of time causes an update to the optimal policy of one or more RL agents, see Equation 5. This is in contrast to model-free Q-learning, where agents only adapt their behaviour after executing actions. 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 optimal 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. As a result of the different feedback models in CRL, agents can utilise more information when learning an optimal policy in a distributed system. Collaborative feedback also enables agents to learn from their neighbours to solve collective system problems.

8 Reducing the Bullwhip Effect in SME Supply Chains using CRL

The “bullwhip effect”, identified by Hau Lee [57], occurs when demand order variability in the supply chain is amplified as orders move up the supply chain. It has direct cost implications for SMEs as excess and insufficient inventories reduce profit levels and efficiencies of companies.

The bullwhip effect can be explained by the famous Procter & Gamble example, where logistics executives examined the order patterns for baby diapers, Pampers [57]. The sales at the retail stores were only slightly fluctuating over time and the consumption was almost constant. At the distributors level the fluctuations of orders were much higher, exceeding surprisingly the variability of the retail sales. However, at the suppliers’ level the orders for materials had even higher fluctuations, with amplitude greater then at the two lower levels.

A similar effect was observed in supply chains of numerous other companies. For example in Hewlett-Packard (HP) the executives discovered that the sales of one of their printers at resellers had moderate fluctuations, while the orders from the resellers had much bigger swings [57]. At the same time, the orders from the printer division to the company's integrated circuit (higher level in the supply chain) had the highest fluctuations.

This phenomenon was called the “bullwhip effect” for its similarity to the swings variability when a whip is cracked – the closer to the end of the whip the greater the swings. High fluctuations and instability of supply chains can lead to serious inefficiencies in a company according to Lee [57], such as excessive inventory investment, poor customer service due to unavailable products or long backlogs, lost revenues, misguided capacity plans, inactive transportation, and missed production schedules.

Prof. Lee shows several major causes of the bullwhip effect. The most important among them, which has the largest contribution to the effect, is demand forecast updating. In every company the managers estimate the expected demand for the company's products in order to maintain the optimal inventory level. There are multiple ways of demand forecasting, most of them are based on extrapolating the history of orders from the immediate customers, for example by exponential smoothing. The forecast is used next to calculate the orders for the supplier, the upstream company in the supply chain. The source of the problem is that in most traditional companies the demand forecasting is performed independently at all levels of the supply chain, using only the information available from the immediate customers. This way imprecise estimations are passed upstream and used as the input for the next estimations, the errors are propagated and magnified upstream. Other causes of the bullwhip effect include order batching, price fluctuations and shortage gaming.

Several remedies to the bullwhip effect have been proposed. For example Lee suggests that the demand data at a downstream company should be available to the upstream company, which would allow all companies in the supply chain to perform their forecasts with the same input data [59]. This strategy has been already applied by IBM, HP and Apple who require sell-through data as part of their contract with the resellers. However, this approach is not possible for SMEs that act as suppliers to larger companies.

There are numerous companies specialized in supply chain management and optimization. For example Rapt released software computing the optimal stock level for each product in the

inventory of a company and assisting in calculating new orders. Sun Microsystems reports that using Rapt's software they saved $15 million and increased revenue by $30 million. SeeCommerce developed supply chain monitoring tools which helped DaimlerChrysler to save $7.2 million in reduced stock and $10 million through improved order fulfilment. Vigilance and PowerMarket are other examples of successful companies in the supply chain management field.

8.1 Analysis

One of the best known theoretical models of the supply chain is the Beer Distribution Game [58] created by John Sterman from MIT. It was thought as a tool for demonstrating the dynamics of an inventory management system for students of economics. Soon it became recognized by the scientists as a powerful model of a supply chain.

It is formulated as a game where four players (who represent four companies: factory, distributor, wholesaler and retailer) distribute beer from the factory to the customer. They act independently without communicating with each other. This strongly resembles the interaction model of many SMEs in a supply chain. Each turn of the game the players deliver the beer stock downstream and send orders upstream. The goal is to minimize the company's cost by optimizing the stock level – the companies have to pay for both stored beer and for outstanding orders. Keeping the stock level low above zero guarantees the minimal cost. The players regulate the inventory level by placing higher or lower orders to the suppliers.

Despite simple rules Sterman shows that the game usually exhibits very complex, unpredictable patterns characterized by instability and high amplitude fluctuations – the bullwhip effect. After performing 48 trials of the game with the students he reports the average human strategy is about 10 times worse then optimal [58]. Moreover, he believes that the oscillations observed in the game can be seen in real-life large production-distribution systems.

There are multiple computer implementations of the beer game [58, 59]. For our own experiments, described later in this chapter, we developed two separate implementations of our model. The first implementation was created in RePast [62], a Java-based framework for multi-agent simulations. The simulations were used for analysis and testing of our algorithms. Figure 18 shows a screenshot of a single supply chain and a plot illustrating the bullwhip effect.

The second implementation was based on web services and Apache Tomcat / Axis [53, 54] platform, with the intention of porting this version to the first implementation of the DBE architecture that will be released in 2005. In particular we are going to integrate the implementation using the Interceptor Architecture to capture data required by the optimisation model and provide service location and leasing information using the Knowledge Base and FADA [39].

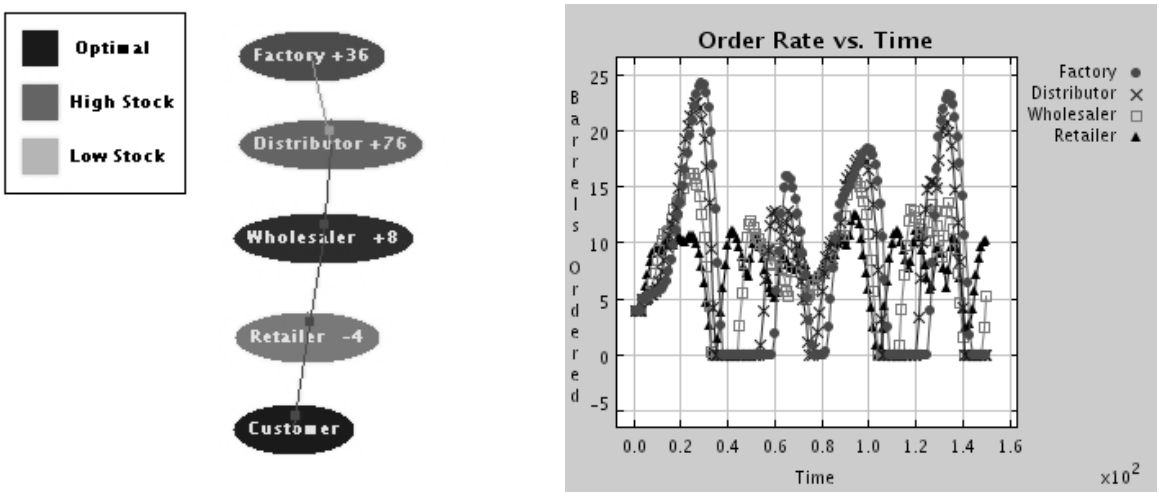


Figure 18 Bullwhip effect

Several methods of obtaining stability in supply chains were proposed [59]. In our model we use adaptive demand expectation according to the following formula:

$$E_t = \theta D_{t-1} + (1 - \theta) E_{t-1}$$

where E_t denotes expected demand in turn t, D_t is the demand in turn t and θ is a constant value between 0 and 1. Orders are calculated according to a strategy called by Sterman "anchoring and adjustment":

$$O_t = max(0,\ E_t + AS_t + ASL_t)$$

where O_t is the order size in turn t, E_t is expected demand in turn t and AS_t and ASL_t are the adjustments to the inventory and to the supply line. The adjustments are meant to correct the discrepancies between the desired stock level (DS) and actual stock level (S) and between the desired supply line (DSL) and the actual supply line (SL), they are calculated from the formula:

$$AS_t = \alpha (DS - S_t)$$
$$ASL_t = \beta (DSL - SL_t)$$

where α and β are constant parameters from (0..1) range. The desired stock level is assumed constant. The current supply line is defined as the amount of beer which was ordered but hasn't been received yet. The desired supply line is determined as the product of the expected demand and the sum of the order delay (ρ) and delivery delay (φ):

$$DSL = E_t (\rho + \varphi)$$

It is shown in [60] that with the correct setting of the parameter values the supply chain can be guaranteed stable, hence the individual costs of each player and the total cost of all players is minimized.

In our research we are interested in scenarios where a single supply chain is replaced by a network of companies (SME) with multiple retailers, wholesalers, factories, etc. Each SME can select one or more supplier and one or more consumers. This way the composition of supply chains is dynamic. Such a situation corresponds better to a real-life market where the

companies can choose and change their business partners according to their strategy and current circumstances. The SMEs optimize their cost not only by controlling the size of orders but also by selecting the best suppliers. Figure 19 presents sample configurations of supply chains and SMEs.

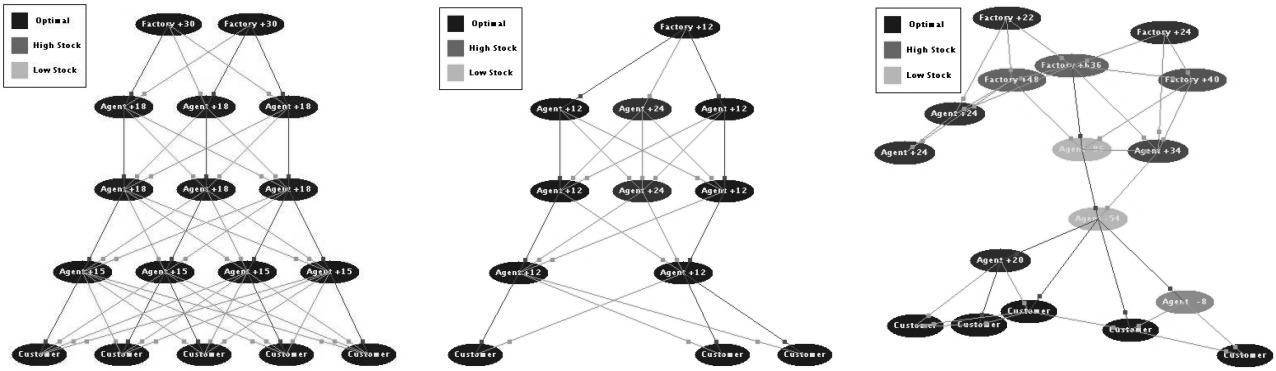


Figure 19 Configurations of the supply chains a) Hierarchical structure b) Hierarchical structure with random changes c) Random graph

None of the companies can possess full information about all suppliers and all consumers. However, the SMEs can cooperate by exchanging and sharing the information. Keeping the supply chains stable is in common interest.

As an optimization technique for decentralized supply chain management we propose Collaborative Reinforcement Learning , see Chapter 7, a meta-heuristic designed for solving optimization problems in decentralized, dynamic and uncertain environments.

In our model we took the following assumptions. Each SME is a separate, independent agent. The neighbours of an agent are its immediate suppliers obtained for example from the semantic registry in the DBE. There is only one state per agent and transitions to external states are deterministic. An agent's goal is to select the best supplier when deciding new order. Each available supplier is a potential action. The actual size of the orders is determined using Sterman's formula discussed above, CRL algorithm is used only for the supplier selection. The actions are selected probabilistically with the Bolzmann distribution [51]:

$$P(a) = \frac{e^{\frac{-Qi(a)}{T}}}{\sum_{a} e^{\frac{-Qi(a)}{T}}}$$

The agents receive a positive reward St for keeping the supply chain stable, and a negative reward B for having backorders. Thus the total reward for choosing a supplier a is:

$$R(a) = St + B$$

In addition, a distance D is defined for any pair of agents. The distance can be computed from the network metrics, such as the latency and bandwidth or from the reliability of the supplier. In the future we plan to use the Interceptor Architecture for this purpose. The distance can also reflect the compatibility of interfaces between the consumer and the supplier obtained from the Semantic Registry or the values calculated by the DBE Recommender.

The Q-learning formula for agent i is defined in the following:

$$Q_i(a) = R_i(a) + (D_i(a) + Decay(V_i(a)))$$

$$V_i(x) = max_a [Q_x(a)]$$

where *a* is a given supplier, *R* is the reward, *D* is the distance to the supplier and *V* is a cached value of the supplier. The agents advertise computed values (*V*) to their customers, cached values are subject to exponential decay:

$$Decay(V) = V \rho^T$$

The initial experiments of our algorithms show that the supply chains in most examined configurations are stable. The agents manage to choose the optimal suppliers according to the distance, the order and delivery delays and the availability of resources. The agents succeed also to place correct orders and keep the inventory low. The individual cost of each SME is therefore minimized. The system adapts to changes, agents arrivals and departures introduce instabilities but the system manages to re-establish a steady state. Sample results are presented in Figures 20 and 21.

We are designing new experiments and developing new metrics to evaluate the performance of our approach more systematically. In particular we are planning to measure the average cost of all agents in the system and the time needed to reach a stable state of the supply chains. With these metrics we can compare the performance our algorithm with other proposed solutions.

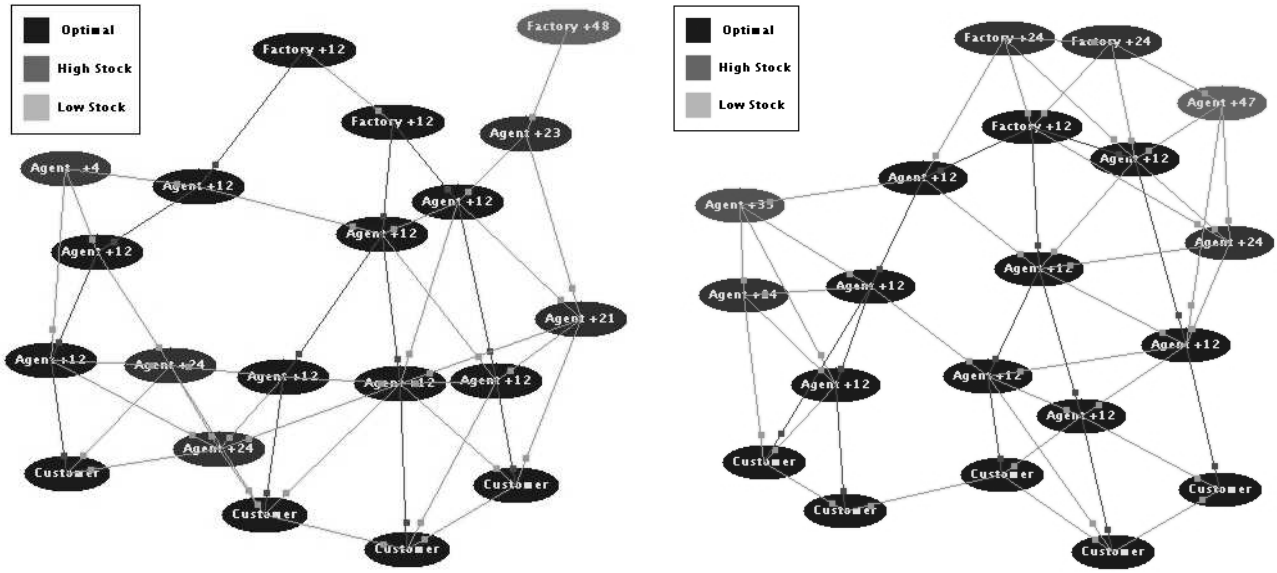


Figure 20 Self-organization of supply chains

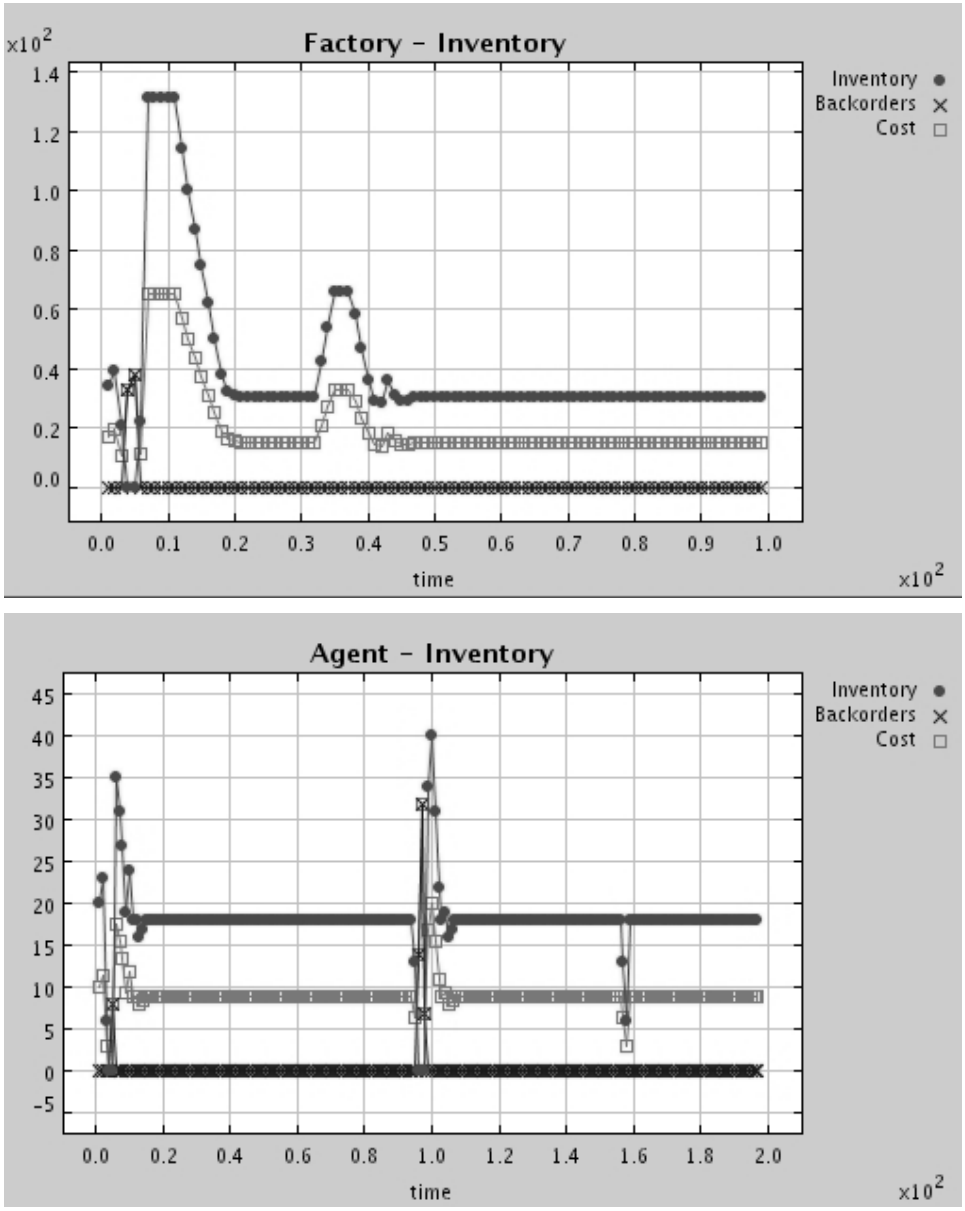


Figure 21 Sample results, the supply chains are stable

8.2 Instructions for how to Run the Simulations using Repast

The simulation has been implemented in Java language and is available as a JAR file. The only requirement to run the program is a working Java virtual machine. The code can be downloaded from Collabnet at:

`src/demos/BeerGame`

The simulation can be started with the following command:

>java –jar BeerGame.jar

9 Conclusion

This document has described preliminary work on the implementation of the EvE. It provided background on the role of the EvE in the DBE and describe three use cases for problem solving using the EvE. An interceptor architecture that is part of the DBE Execution Environment is defined that sets up a link between the EvE world where optimization can be performed and the DBE execution environment and system, where DBE services, providers and users reside.

Background research on self-organising distributed systems is presented, and from this a set of requirements are established for decentralised optimisation techniques, such as a decentralised EvE. The design of a decentralised optimisation technique, called collaborative reinforcement learning, is then presented and its application to one of the three use cases for intelligent tools in the DBE is then evaluated.

Finally, a timetable for the implementation of the EvE for the second half of the DBE project is outlined.

10 Bibliography

[1] Jim Dowling, Eoin Curran, Raymond Cunningham and Vinny Cahill, "Collaborative Reinforcement Learning of Autonomic Behaviour", 2nd Workshop on Self-Adaptive and Autonomic Computing Systems, 2004.

[2] Eytan Adar and Bernardo A. Huberman. Free riding on gnutella. *First Monday,* September 2000.

[3] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47-97, Jan 2002.

[4] Robert Axelrod. *The Evolution of Cooperation*. Basic Books, New York, 1984.

[5] Sonja Buchegger and Jean-Yves Le Boudec. Performance analysis of the CONFIDANT protocol: Cooperation of nodes -- fairness in dynamic ad-hoc networks. In *Proceedings of IEEE/ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC)*, Lausanne, CH, June 2002. IEEE.

[6] V. Cahill, E. Gray, J.-M. Seigneur, C. Jensen, Y. Chen, B. Shand, N. Dimmock, A. Twigg, J. Bacon, C. English, W. Wagealla, S. Terzis, P. Nixon, G. Serugendo, C. Bryce, M. Carbone, K. Krukow, and M. Nielsen. Using Trust for Secure Collaboration in Uncertain Environments. In *Pervasive Computing Magazine*, volume 2, pages 52-61. IEEE, 2003.

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

[8] Gianni Di Caro and Marco Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317-365, 1998.

[9] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies*, pages 46-66. Springer-Verlag New York, Inc., 2001.

[10] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[11] Eoin Curran and Jim Dowling , "SAMPLE: Statistical Network Link Modelling in an On-Demand Probabilistic Routing Protocol for Ad Hoc Networks", International Conference on Wireless on Demand Network Systems and Services, 2005.

[12] Marco Dorigo and Gianni Di Caro. The ant colony optimization meta-heuristic. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, pages 11-32. McGraw-Hill, London, 1999.

[13] Gary William Flake. *The computational beauty of nature*. MIT Press, 1998.

[14] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 21-35. Springer-Verlag, 1997.

[15] Márk Jelasity, Wojtek Kowalczyk, and Maarten van Steen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, November 2003.

[16] Márk Jelasity, Wojtek Kowalczyk, and Maarten van Steen. An approach to massively distributed aggregate computing on peer-to-peer networks. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing 2004*, pages 200-207, A Coruna, Spain, February 2004. IEEE Computer Society Press.

[17] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7-38, 1998.

[18] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[19] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41-50, 2003.

[20] Rohit Khare and Richard N. Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. In *Proceedings of the 26th International Conference on Software Engineering*, pages 428-437, Edinburgh, Scotland, United Kingdom, May 2004. IEEE Computer Society.

[21] Alberto Montresor, Hein Meling, and Özalp Babaoglu. *Toward Self-Organizing, Self-Repairing and Resilient Distributed Systems*, chapter 22, pages 119-124. Number 2584 in Lecture Notes in Computer Science. Springer-Verlag, June 2003.

[22] M. Newman. Models of the small world: A review. *Journal of Statistical Physics*, 101(3-4):819-841, Nov 2000.

[23] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 234-244. ACM Press, 1994.

[24] harles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 90. IEEE Computer Society, 1999.

[25] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161-172. ACM Press, 2001.

[26] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164-206, May 2003.

[27] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[28] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329-350. Springer-Verlag, 2001.

[29] Ruud Schoonderwoerd, Owen E. Holland, Janet L. Bruten, and Leon J. M. Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive Behavior*, 5(2):169-207, 1996.

[30] Suresh Singh, Mike Woo, and C. S. Raghavendra. Power-aware routing in mobile ad hoc networks. In *Mobile Computing and Networking*, pages 181-190, 1998.

[31] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149-160, 2001.

[32] Panagiotis Thomas, Demosthenis Teneketzis, and Jeffrey K. Mackie-Mason. A market-based approach to optimal resource allocation in integrated-services connection-oriented networks. *Oper. Res.*, 50(4):603-616, 2002.

[33] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gun Sirer. KARMA: A secure economic framework for p2p resource sharing. In *In Proceedings of the Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, California, June 2003.

[34] Tom De Wolf and Tom Holvoet. Emergence and self-organisation: a statement of similarities and differences. In S. Brueckner, G. Di Marzo Serugendo, A. Karageorgos, and eds Nagpal, R., editors, *Proceedings of the International Workshop on Engineering Self-Organising Applications 2004*, pages 96-110, 2004.
[35] S. Wolfram. Universality and complexity in cellular automata. *Physica D*, 10:1-35, 1984.
[36] F.Nachira, E. Chiozza, H.Ihonen, M.Manzoni, F. Cunningham, Towards a Network of Digital Business Ecosystems, 2002.
[37] Paolo Dini et Al., DBE Technical Annex I, DBE EU Deliverable, 2003.
[38] Sun Microsystems, DBE Execution Environment Architecture, DBE EU Deliverable, 2004.
[39] Soluta, DBE Architecture Requirements, DBE EU Deliverable, Nov 30 2004.
[40] Gerard Briscoe, Jon Rowe, Paolo Dini, Evolutionary Environment Discussion Paper, DBE Internal Document, 2004.
[41] Jim Dowling, Eoin Curran, Raymond Cunningham and Vinny Cahill, "Using Feedback in Collaborative Reinforcement Learning to Adaptively Optimise MANET Routing", IEEE Transactions on Systems, Man and Cybernetics (Part A), Special Issue on Engineering Self-Organized Distributed Systems, March '05.
[42] Jim Dowling and Vinny Cahill, "Self-Managed Decentralised Systems using K-Components and Collaborative Reinforcement Learning", Proceedings of the Workshop on Self-Managed Systems (WOSS'04), 2004.
[43] Jim Dowling, Eoin Curran, Raymond Cunningham and Vinny Cahill, "Collaborative Reinforcement Learning of Autonomic Behaviour", 2nd Workshop on Self-Adaptive and Autonomic Computing Systems, 2004.
[44] Jim Dowling, Eoin Curran, Raymond Cunningham and Vinny Cahill, "Component and System-Wide Self-* Properties in Decentralised Distributed Systems", Workshop on Self-* Properties in Complex Information Systems, 2004.
[45] Ilya Prigogine, Order Out of Chaos, 1984.
[46] A.E. Eiben and J.E. Smith, Introduction to Evolutionary Computing, Springer, 2003, ISBN 3-540-40184-9.
[47] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, 1998.
[48] R.E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
[49] C. Watkins. Learning from delayed rewards. *PhD Thesis, King's College, Cambridge*, 1989.
[50] M. Littman and J. Boyan. A distributed reinforcement learning scheme for network routing. *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications*, pages 45-51, 1993.
[51] L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237-285, 1996.
[52] Paulo Dini, "Report on DBE-Specific Use Cases", DBE Deliverable 18.1, 2004.
[53] The Apache Foundation, Tomcat, http://jakarta.apache.org/tomcat.
[54] The Apache Foundation, Apache Axis, http://ws.apache.org/axis.
[55] CVSSP, SEPS, Univ. of Surrey, Optimisation – Model Adopted, DBE Deliverable 12.1, 2004.
[56] Jonathan Rowe, Distributed Evolutionary Computing Architectures, DBE Mozart Workshop at STU, July 2004.
[57] Hau L Lee, V Padmanabhan and Seungjin Whang, "The bullwhip Effect In Supply Chains", In *Sloan Management Review,* 38(3):93-102, Spring 1997
[58] John D Sterman "Modeling managerial behavior: Misperceptions of feedback in a dynamic decision making experiment", *Management Science*, 35(3):321-339, 1989
[59] Owen Densmore "The Emergence of Stability in Diverse Supply Chains", In *Proceedings of the International Workshop on Self-* Properties in Complex Information Systems*, Bologna, Italy, May 2004

[60] Michael North and Charles Macal "The Beer Dock: Three and a Half Implementations of the Beer Distribution Game", *Sixth Annual Swarm Users Meeting (SwarmFest'02),* Seattle, Washington, March 2002
[61] The Supply Chain Simulation Workgroup, http://complexityworkshop.com/sun/SCSim/
[62] RePast, The Recursive Porous Agent Simulation Toolkit, http://repast.sourceforge.net/

Appendix A - Testing the Interceptor Demo

The Interceptor Demonstrator using handlers

Before setting up the interceptor demo it is necessary to have a running installation of:

- Apache Tomcat 5.0.x
- Apache Axis 1.x
- MySQL 4.x

Accessing code via CVS using the command line

```
>cvs login
>cvs co src/demos/interceptors-client
>cvs co src/demos/interceptors-server
```

Building the code using maven

```
>cd src/demos/interceptors-client
>maven site && maven eclipse

>cd src/demos/interceptors-server
>maven site && maven eclipse
```

The above commands will check out the code (both client and server side interceptors) from CVS, compile them (via maven) and generate eclipse project files (via maven).

Setting up MySQL

Before setting up the actual service it is necessary to create a number of tables in the MySql database. This is done by using the SQL scripts in:

```
src/demos/interceptors-server/src/etc
```

Use the MySQL command line utilities or a MySQL GUI to do this. Once the tables are set up axis needs the JDBC drivers for MySQL placed on its class path. This is best done by placing the MySQL JDBC driver jar file in `axis/WEB-INF/lib.`

Working with the code

Open eclipse and import the projects. The client application will run straight from eclipse (once the service and service interceptors are deployed) via the run menu.

Deploying the demo

To deploy the service and service interceptors:
1. Firstly check the configuration of both

`src/demos/interceptors-server/src/wsdl/deploy.wsdd` and `src/demos/interceptors-client/src/bin/client-config.wsdd.`

2. Copy the contents of `src/demos/interceptors-client/src/bin` to `axis/WEB-INF/classes`
3. Export the interceptor-server project as a jar file (`maven jar:install`) and place the generated files in `axis/WEB-INF/lib`

From the command line:

```
java -cp $AXIS_LIBS org.apache.axis.client.AdminClient
src/demos/interceptors-server/src/wsdl/deploy.wsdd
```

This will register the deployed service with the axis engine along with the custom handlers.

It will be now possible to invoke the web service that uses the custom handlers.

Acronyms

AI = Artificial Intelligence
CA= Cellular Automata
CRL = Collaborative Reinforcement Learning
DBE = Digital Business Ecosystem
DOP = Discrete Optimisation Problem
EA = Evolutionary Algorithm
EC = Evolutionary Computing
EvE = Evolutionary Environment
FL = Fitness Landscape
MANET = Mobile Ad-Hoc Networks
MDP = Markov Decision Process
P2P = Peer-to-Peer
SM = Service Manifest
SME = Small and Medium Sized Enterprises
WAN = Wide Area Network