Distributed Artificial Life Toolkit

project report

Vlad-Cătălin Mereuţă

Project supervisor: Dr. Paul Goldberg

2001-2002

Abstract

This paper covers the design and usage of a toolkit (DALT) for building distributed artificial life simulations.

A review of the background material is given, followed by a synopsis of the different technologies used in the construction of DALT. The auxiliary tools and libraries used are covered an overview of the toolkit design is given. The paper continues by detailing the main implementation issues.

DALT has been used to build a *Game of life* simulation, which allowed for a series of experiments to be ran. The results obtained indicate that using DALT can give a significant performance gain for computationally intensive artificial life simulations.

Keywords

multi-agent system — distributed computation — artificial life — SOAP — C++ library

Acknowledgements

Eric D. Cohen, the author of *ZThreads* library (http://zthreads.sf.net) provided support in tracking down and fixing a series of thread-related issues, both with DALT and *ZThreads*.

Contents

List of Figures

Chapter 1

Introduction

1.1 Project resources

This project has its own web-site: http://dalt.sf.net, hosted by *SourceForge*. The project website holds the latest version of the source code (straight from CVS or packaged). The website also provides access to the design document, the specification and the complete API documentation for the class hierarchy.

SourceForge provides discussion groups and a bug tracking mechanism for the project.

1.2 Motivation

In the recent years the number of experiments related to *Artificial Life (alife)* has increased dramatically, involving many areas unrelated to computer science. Many of these projects share a common design. For example, there are many predator-prey simulations, each having slightly different requirements and goals. Most of the simulations involve some form of environment (usually 2-dimensional) together with a number of various types of entities, usually competing for some resource(s), moving and interacting with each other and their environment.

It became obvious that packaging the common features together in a re-usable form would greatly decrease the amount of implementation work required for running Artificial Life experiments. This gap was addressed relatively quickly, and now there are a few functional toolkits for agent-based simulations. Probably the most complete and well established is the *SWARM* toolkit

(see section 1.6.2.2 on page 23 for more details).

The problem with these libraries is that they do not scale well to big experiments, ignoring the combined processing power which can be gathered from several networked workstations, or even the Internet. Many of the experiments are computationally intensive [Doran and Gilbert, 1994, Servat et al., 1998]. For example, the now famous *EOS project* [Doran et al., 1994] involved a large number of complex agents, each training an internal neural network at every step of the simulation as well as following a planning algorithm in order to decide on the next action(s). For experiments such as these, where a large number of agents is required, gathering meaningful results by repeated and extensive runs of the same simulation can be a time and resource consuming task. It is usual that experiments of this kind are usually repeated several times in order to ensure that the representative results have been recorded. The only thing to do in these cases is to attempt to distribute the computationally intensive task of agent processing over several physical processing units, which can be either separate processors in the same computer, or distinct networked computers.

The *Distributed Artificial Life Toolkit (DALT)* attempts to address the issue described above by providing a set of software tools that can be used to facilitate the construction of distributed agent-based artificial life simulations.

It is worth mentioning that most experiments dealing with social simulation are good candidates for being implemented using DALT. It has been shown [Conte et al., 1998] that social simulation experiments would greatly benefit from using an agent based approach. The features that will be provided by DALT (see following section) can be put to full use by such simulations.

This project shall not aim to create yet another multi-agent system. Most agent based systems implement one/more agent communication systems together with a strong representation of the agents themselves, both as internal and external models. While this can be a good thing if particular experiments happen to fit exactly within the boundaries of the system, it can mean that a lot of extra work has to be done in order to adapt existing systems to specific experiment requirements.

Most artificial life simulations are suitable for discrete simulation. The multi-agent systems lack the synchronisation features needed by such an approach, concentrated simply on providing an event based framework. Artificial life experiments usually require the simulations to run in cycles,

with a well-defined procedure. Ignoring this feature would render the task of obtaining maximum efficiency very difficult.

Most agent-based systems will assume agents as being (mostly) independent and self-sustained. This is simply not practical in simulation with a large number of agents due to the difficulty of dealing with synchronisation issues (many agents can attempts to use the same resource at the same time). Having the agents as completely independent entities can also mean that extracting data from the simulation itself can be difficult, as mechanisms have to be constructed to centralise and interpret data coming from each individual agent.

Out of all the existing multi-agent systems only a small number are distributed. The distributed-agent based systems are not entirely suitable for artificial life simulations. Some of the extra difficulties introduced by distributed multi-agents systems are related to their model of agents. Agents are usually seen as separate programs, which would very seldom share the same machine. A lot of work has been done in the mobile agent area. While mobile agents can be useful in some circumstances this approach would not scale very easily to environments with hundreds of agents.

All the issues raised above will be addressed by DALT.

1.3 Objectives

In its final form the *Distributed Artificial Life Toolkit* will be composed of two libraries, a tool for generating simulation environments and a tool for monitoring and manipulating the running of simulations. One of the libraries will be providing classes that can be used to build and manage the simulation environment and the other will provide classes that can be used to build agents and to describe agent interaction.

The project will come complete with one or two example simulations which should provide a start point for anyone wanting to write their own simulation.

By using DALT, implementing a distributed alife application should require a minimum effort for implementing and managing distributed processing.

DALT does not aim to elaborate on agent architecture. Different projects will have widely different requirements. By providing a flexible skeleton for these kinds of applications, the toolkit

should be usable in a wide range of simulations, with different requirements.

When the toolkit is complete, a computationally intensive simulation based on DALT should run faster when distributed across several physical machines then when ran on a single machine.

The toolkit should be language independent and modular. It should be possible to re-implement parts of the toolkit in different programming language without affecting the other modules. This should enable experiments which require a very specific programming language to use the toolkit.

The implementation of the toolkit should be portable, as much as possible. The platform independence is very important, as simulations are likely to run on networks with mixed hardware and operating systems configurations.

DALT is at its core a toolkit for distributed computation. However the project aims for something a lot more specific then a distributed computation toolkit. DALT will provide support for agents, a model for constructing simulations in a computationally efficient way, means for synchronisation and front-end tools and specifications. None of the distributed computation toolkits offer these features.

1.4 Results

Once implemented, the toolkit has been used to create a “Game of life” simulation which enabled testing the performance of the toolkit in a series of experiments.

The tests revealed significant performance improvements in computationally intensive simulations (up to 45%) when distributed. Simulations which are not computationally intensive suffer a slight degradation in performance as a result of being distributed.

1.5 Conclusion

It is possible to speed up artificial life simulations by distributing the processing required for the agent across a network of workstations (NOW). DALT supplies the means and methodology for designing and implemented such distributed simulations.

However, as different simulations can have very different requirements, it is impossible to provide a comprehensive solution perfectly suited for any type of artificial life simulation without

the need for modifying some of the basic elements of the toolkit.

DALT covers most of the foreseeable problems, providing a flexible framework which can be easily adapted to cope with most unforeseen issues.

1.6 Background material

The following sections will cover some background material relevant to the scope of this paper.

1.6.1 Distributed systems

A distributed system has three primary characteristics [Mullender, 1994]:

multiple computers it contains more then one physical computers

interconnections some of the I/O paths will interconnect the computers

shared state the computers cooperate to maintain some shared state

The aspect which concerns this project the most is the communication method which is to be used for the systems to interact with each other. Below is a quick review of some of the most commonly used packages for distributed computation, along with reasons for not using them for this particular project[1]. Choosing a certain communication method for distributing a piece of software can greatly influence the design of that software. This is the main reason for which the methods below reviewed.

Refer to section 1.6.1.4 on page 17 for a discussion of the communication method which was finally chosen.

1.6.1.1 Parallel Virtual Machine (PVM)

PVM is a public domain message-passing system which has been ported on most systems. It features a reasonable list of abilities, including *Remote Method Invocation (RMI)* (blocking, non-blocking and timed), broadcasting and multi-casting. Unfortunately it only supports C, C++ and Fortran as programming languages. The implementations available for Linux do not support C++

[1] this bears a direct relation to the project objectives, as they are described in section 1.3 on page 13

directly – they just wrap around the C interfaces. This is a big problem with most of the RMI tools. In object orientated programming objects are often arguments for methods and quite often, the values returned by methods are objects.

The procedural methods to not map well to object orientated architectures, requiring the programmer to put in extra effort in order to circumvent the lack of representation for objects.

1.6.1.2 Java RMI

The Java RMI package provides excellent functionality [see SUN]. However, it only works with Java, and Java may not be the language preferred by many developers. An inherent constraint is that all the components of the toolkit which need to communicate with each other will have to be written in Java. This is not acceptable, as one of the goals of the project is to maintain modularity and language independence.

1.6.1.3 DCOM and CORBA

DCOM and CORBA are two of the most popular protocols for distributed computing. They provide object-orientated RMI, and they are language independent. However, the implementations for both protocols are big. They require a large amount of memory and extra processing power. The specifications are complex resulting in a very steep learning curve, even for experienced developers

Custom object marshaling is still required for many applications and this can introduce errors (mostly related to byte-ordering and the same data types being slightly different on different machines). Another issue with DCOM and CORBA is that the debugging process can be extremely difficult and time consuming. Many applications grew their own ASCII-based communication protocols in parallel with using DCOM/CORBA in order to successfully debug their code.

Even more, the binary contents of the message represents a security risk, as messages are difficult to filter by network administrators. Also most RMI methods tend to dynamically allocate communication ports, which increases the amount of effort required in order to secure a network which uses DCOM/CORBA.

1.6.1.4 SOAP

SOAP [W3C, 2001] is the communication protocol which was chosen for this library. Below is a brief overview of the protocol, together with the main reasons which lead to choosing it.

1.6.1.4.1 Overview *Simple Object Access Protocol (SOAP)* defines an XML based communication format. Since its conception every major software developer announced its support [Seely, 2002].

It contrasts the other protocols by being light-weight and ASCII-based. It can be implemented on nearly any device and the ASCII encoding of the messages makes it really easy to debug. Unlike most of the other protocols, SOAP does not attempt to define its own communication protocol; it uses existing protocols instead.

Initially, SOAP was specified over HTTP, but more recent specifications cover SMTP, FTP and other mediums. The advantage of this approach is that SOAP can be easily integrated with the already existing base of applications and protocols.

The fact that SOAP is XML based means that any programming language that can parse XML and has networking support can be used for SOAP. This is a huge advantage, as it greatly increases the portability of the protocol without the need of stub compilers (as DCOM, CORBA an JAVA RMI require).

To make implementations even easier, SOAP is using only a cut-down version of XML. For example features, of XML like Document Type Definitions (DTDs) [Refsnes, DuCharme, 1999] and Processing Instructions [DuCharme, 1999] are not allowed in SOAP XML documents.

In contrast with other implementations (such as CORBA), SOAP sacrifices some features (distributed garbage collection, batching of messages, object-by-reference, activation) for simplicity and speed. However, SOAP is a evolving protocol so these features (or equivalent ones) might eventually make their way in the specification, possibly as optional sections.

Another important feature provided by SOAP is security. The communication channels can be encrypted making the message transfer secure.

1.6.1.4.2 XML Describing the XML format is beyond the scope of this paper. Good references can be found on the web (`http://www.w3.org`) or in paper format [DuCharme, 1999].

1.6.1.4.3 SOAP specification SOAP specifies the following items: [Seely, 2002]

- a packaging model (the SOAP envelope)
- a serialisation mechanism (the SOAP encoding)
- a RPC mechanism (the SOAP RPC representation)

These different pieces of the protocol can be used in any combinations. They can even be used independently from each other.

The serialisation mechanisms specifies a set of strict rules for encoding the following data types: values (can be string, number,date, enumeration or composite of many primitive values), simple values (no named parts), compound values (named parts), accessor (key for retrieving fields in compound values), array (compound value with accessor being the ordinal position of each element), struct (accessor name used to distinguish between fields), simple type and compound types (defined using XML schema or an array).

For example here is a sample request/response communication, used for creating an agent:

SOAP message (request)

```
REQUEST:
POST / HTTP/1.1
Host: ra.deity:8001
User-Agent: EasySoap++/0.6
Content-Type: text/xml; charset="UTF-8"
SOAPAction: "http://dalt#createAgent"
Content-Length: 432

<E:Envelope xmlns:E="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:A="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:s="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:y="http://www.w3.org/2001/XMLSchema"
 E:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<E:Body>
    <m:createAgent xmlns:m="http://dalt">
        <agent_id s:type="y:int">1</agent_id>
        <type_id  s:type="y:string">cell</type_id>
```

```
19      </m:createAgent>
20  </E:Body>
21  </E:Envelope>
```
SOAP message (request)

This is an example of a call to the method `createAgent` (line 6,16) on the machine `ra.deity:8000` (line 3). The method has two parameters: `agent_id` (integer) on line 17 and `type_id` (string) on line 18. The `http://dalt` namespace is used to specify all XML attributes.

SOAP message (response)
```
1   RESPONSE:
2   HTTP/1.1 200 OK
3   Content-type: text/xml; charset="UTF-8"
4   Content-Length: 409
5   Connection: close
6   Date: Fri, 08 Feb 2002 18:51:46 GMT
7
8   <E:Envelope xmlns:E="http://schemas.xmlsoap.org/soap/envelope/"
9               xmlns:A="http://schemas.xmlsoap.org/soap/encoding/"
10              xmlns:s="http://www.w3.org/2001/XMLSchema-instance"
11              xmlns:y="http://www.w3.org/2001/XMLSchema"
12   E:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
13
14  <E:Body>
15      <m:createAgentResponse xmlns:m="http://dalt">
16          <performed s:type="y:int">0</performed>
17      </m:createAgentResponse>
18  </E:Body>
19  </E:Envelope>
```
SOAP message (response)

In the response to the method call described above, another call is returned, `createAgentResponse`. The SOAP layer intercepts this call and filters it out returning only the result (`performed`) to the caller.

1.6.1.4.4 Implementations This project makes use of three different SOAP implementations, for three different languages: C++, Java and Python. Please refer to section 2.1.1.4 on page 28 for details on particular SOAP implementations.

1.6.2 Agent based simulation and artificial life

A widely accepted definition[2] for Artificial Life is given by Christopher Langsten, one of the main developers behind SWARM:

> Artificial Life ("AL" or "Alife") is the name given to a new discipline that studies "natural" life by attempting to recreate biological phenomena from scratch within computers and other "artificial" media. Alife complements the traditional analytic approach of traditional biology with a synthetic approach in which, rather than studying biological phenomena by taking apart living organisms to see how they work, one attempts to put together systems that behave like living organisms.

There has always been a strong link between the distributed computation field and the artificial intelligence field. Initially most of the distributed applications were presented at various artificial intelligence conferences. Eventually, the distributed computation field branched, roughly following a principle stating that as soon as a distributed application reaches the stage at which the time necessary for contacting other subsystems is not negligible, the application stops being classified as "Parallel Problem Solving" and starts being classified as "Distributed Artificial Intelligence (DAI)" [Tokoro, 1996].

In distributed systems it can be the case that different subsystems have the notion of "self". The subsystems can also begin to exhibit different characteristics. This is the main feature specialising DAI into *Multi-Agent Systems (MAS)*.

Most of the alife simulations are suitable for an agent-oriented approach (see section 1.6.2.1 on the facing page for information on *agents*). This is also true for simulations concerning social phenomena, which are usually a specialised subset of the alife simulations.

Most of the experiments based on *cognitive* agents would be suitable for being distributed, as each agent requires a fair amount of computing power. The *reactive* agents on the other hand would not distribute easily. While simulations using reactive agents can still be implemented using the toolkit (as they follow the general architecture described above) the speed limit on the simulation would be imposed by the external events (i.e. communication, perception, actions) as opposed to internal agent processing. The fact that all of the communication travels over a

[2] the definition can be found at http://www.alife.org together with a longer review of the alife field

network that is significantly slower than the speeds achieved internally by computers, would only slow the simulation even more. This issue is illustrated in chapter 3 on page 73.

It has been shown [Werner, 1996] that purely reactive agents cannot scale to large systems. The apparently "magic" way in which simple reactive agents solve their problems lies in either the way their programs are constructed, or in the environment itself. As agent architectures move away from a purely reactive approach, more and more computational complexity is introduced. Some of the usual computationally intensive approaches involve planning [Handel et al., 1996], neural networks, [Doran et al., 1994] and/or other processor intensive methods. In these cases the computational power becomes an important factor in the success of the simulation.

It is worth mentioning that many of the recent artificial life simulations take place within the field of social simulation, which concentrates mainly on the interactions between different agents. Most of these simulations are based on cognitive agents, usually requiring significant processing power.

1.6.2.1 Agents

There is no precise definition of an "agent", as the term is used in a vague way. However, the following common characteristics can be identified [Ferber, 1999]:

Definition 1 *An **agent** is a physical or virtual entity with the following properties:*

(i) it is capable of acting in an environment

(ii) it can communicate directly with other agents

(iii) it is capable of perceiving its environment (but to a limited extent)

(iv) is driven by a set of tendencies (in the form of individual objectives or a satisfactory

(v) it possesses resources of its own

(vi) it has only a partial representation of its environment (and perhaps none at all)

(vii) it possesses skills and can offer services

(viii) it may be able to reproduce itself

(ix) its behaviour tends toward satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representations and the communications it receives.

We shall consider agents at their highest abstraction level[Russell and Norvig, 1995]. That is, we shall only consider the implementation of the first three items in Definition 1 on the preceding page. In order to achieve this we need to provide tools for implementing the following:

- agents performing actions in their environment
- sensory perception
- communication

The rest of the points are directly related to *internal* agent behaviour and will differ widely in different experiments. A point worth mentioning is that *communication* is really just a special, combined, case of perception and action.

Two main approaches to agent design can be distinguished. The first one is building reactive agents. The agents built solely by using the reactive method will react to their environment using a set of simple rules.

The second approach is the cognitive one, in which the agents act as a result of more complex inference process. The perceptions are still used as a starting point, but the agents do not simply respond to their senses. Cognitive agents usually have their own goals and possess planning capabilities.

There are many examples in which hybrid approaches were taken, in an attempt to combine the reflexive approach with the cognitive one. One of the most interesting (and unfortunately unfinished cases) is described in Brooks [1992]. A critique of this example can be found in Werner [1996].

Note that all agents using a cognitive approach will require significantly more processing power then the reactive agents.

1.6.2.2 SWARM

SWARM[3] is a toolkit which can be used for running agent-based simulation. SWARM was originally developed at the Santa Fe Institute in 1994. Now it is maintained by an independent group.

SWARM provides functionality like task schedule management, memory management, GUI widgets, random number generators, and a collections library. It has grown to be the most widely used piece of software of its kind.

However, there are some shortcomings, some of which have been presented (together with counter-arguments) in Lancaster[4]. The most relevant one (to DALT) is that SWARM would not scale properly in a distributed environment. The authors maintain that development work is underway to address this issue by executing certain regions of an agent's code by using a distributed computation toolkit such as PVM (see section 1.6.1.1 on page 15). However the main issue still remains: SWARM is not designed from the ground up with a distributed environment in mind. It is very likely that an extension for distributed processing will be added eventually but it will probably not fully integrate with the rest of the SWARM tools, unless the entire toolkit would be re-engineered.

[3] http://www.swarm.org

[4] Alex Lancaster was a former developer of SWARM

Chapter 2

DALT Methodology

The first section of this chapter will present all the external[1] tools and libraries used by DALT. The paper continues by detailing the architecture and implementation of the C++ libraries used in DALT throughout section 2.3 on page 39 after giving an overview of the top-level architecture of DALT in section 2.2 on page 35.

Section 2.4 on page 63 shall deal with the design and implementations of the tools provided by DALT. This chapter concludes with a description of the process that should be followed to create an alife simulation based on DALT.

The API documentation is not reproduced in this paper due to its large size but it is made available online, on the project website. The API documentation contains full documentation for classes, methods and attributes, together with class inheritance and class dependency graphs for each class of the project. File dependency diagrams are also available.

The most recent version of the source code for the entire library can be browsed online (syntax highlighted) at this address: http://cvs.sf.net/cgi-bin/viewcvs.cgi/dalt/dalt.

2.1 DALT dependencies

2.1.1 Tools used by the project

This section shall review all the main tools and libraries used by the program, together with arguments for using them and examples, where appropriate. In order to reproduce the experiments

[1] outside of the libraries supplied with the programming language

described below you will need all these tools to be installed on the machine you are carrying the experiments on.

This project has quite a high number of external dependencies. This is due to the fact that I have tried to make the final software product as stable and feature-full as possible in the available time. Re-writing software, libraries and tools that have been implemented already is time consuming and the end-result is likely to have less features and perform worse.

Another benefit gained from using the libraries and tools detailed below is that they increase the portability of the toolkit. All the features which are greatly machine-dependent (such as threading, real time clock management, XML parsing, TCP/IP communication, etc) are handled by libraries which have been chosen to be as portable as possible. As the libraries themselves run on several different platforms, this ensures that the toolkit would run cleanly, hopefully with no changes to the code on all the platforms supported by the libraries.

The reasoning behind the language choices for this project will be explained later on in this chapter.

2.1.1.1 C++ STL

The C++ *Standard Template Library* [Stroustrup, 1997, Eckel, 2000] provides many classes for manipulating strings, lists, vectors, queues, hashes and other data structures. It also provides a wide range of generic algorithms (such as set operations, sorting, etc.). The DALT makes heavy use of many of these features in most of its internal components

I am confident that using the *STL* greatly decreased the development type and increased the readability and portability of the code.

2.1.1.2 Autotools

Developing large projects with many dependencies on Unix platforms is not straight forward. This task is made even more difficult when the code has to be portable.

Fortunately, the *GNU Autotools* family can be of great help [Gkioulekas, 1999, Vaughan et al., 2000]. The following applications are part of the family:

autoconf applications and scripts that can be used to generate system-dependent configura-

tion files. This can be used to compensate automatically for missing functions in system libraries, header files named differently, missing libraries (or libraries that are not recent enough) and many other compatibility issues;

automake set of tools which help the developer to generate `Makefile`s compatible with the GNU coding standards. The `automake` tools provide `Makefile`s with a set of default targets (such as `clean, install`) and help with the task of analysing object file dependencies;

libtool a set of applications which help standardise the generation of static and shared libraries across different platforms. This tools are very useful, as different systems will behave differently when required to construct shared and static libraries;

other there are a few other tools (such as `autoheader`) which are not use directly, but rather indirectly by being relied upon by the three main modules described above.

The *Autotools* suite is based mostly on the *M4* macro processing language.

Using *Autotools* for DALT involved a rather large set-up overhead. The tools have a steep learning curve as and the documentation is sparse. The manuals provided with the distribution (in `texinfo` format) serve as reference for each separate tool in the package, but do not give an overview of the whole system and the interactions between different modules. The best source of external information is probably Vaughan et al. [2000]. Once familiar with the tools there is the overhead of modifying the existing code and altering the structure of the source tree to take advantage of the benefits of *Autotools*.

However, these downsides are compensated by increased portability (which is vital for this project) and decreased compilation times, due to the dependency management. Using *Autotools* also provided an easy to use compile interface. All that is required in order to configure the project for a machine, check if the relevant libraries can exist and compile the project is two commands:

```
./configure
make
```

The `configure` command will check the machine configuration and warn the user if there are missing library dependencies. This command creates a `config.h` file containing project-level macro definitions and macro conditions.

For this project, two configuration scripts have been written. The configuration scripts are to processed with `autoconf` and make use of a few custom *M4* scripts. They serve the following objectives:

- enabling and disabling debug information via `configure` command line parameters
- enabling and disabling profile information via `configure` command line parameters
- setting up the C++ compiler and verifying the existing libraries
- adding the libraries and the header files to the compiler command line

Also a set of `Makefile.am` files have been written to be processed with `automake`. These files specify which components of the projects are to be installed after the compilation, what are the main project modules and how to build shared libraries.

2.1.1.3 Doxygen and Graphwiz

Doxygen[2] is a tool used for generating API documentation from C, C++ and Java source code. *Graphviz*[3] is a tool for plotting graphs. If both tools are present, the *Doxygen*configuration files supplied with the project can be used to generate complete API documentation, together with class inheritance and dependency diagrams.

While this is not vital for the project, as the end result is a shared library having a quick reference for the various features provided, together with instructions on how to use them can be extremely helpful.

2.1.1.4 SOAP implementations

There are many implementations of SOAP for several languages and platforms[4]. However, it proved difficult to find a lightweight, reliable implementation. It is even more difficult to find an implementation that offers statefull implementation for SOAP web services. The reason for this is that SOAP was initially design as a messaging protocol. While it is inevitable that many statefull

[2] http://www.stack.nl/~dimitri/doxygen

[3] http://www.graphviz.org/

[4] see http://www.soaprpc.com/software/ for a comprehensive reference

implementations will appear eventually [Cohen, 2001] at the moment this type of implementations is sparse.

2.1.1.4.1 EasySoap++ *EasySoap++*[5] is a C++ library handling the SOAP protocol. It is one of the very few libraries which handles statefull communication. This is accomplished by using a slightly modified version of the *Abyss*[6] web server. *Abyss* is an extremely light-weight and very quick server, designed to have a minimum memory footprint and maximum speed.

The usual approach is to have an external web server/service call arbitrary functions within the SOAP server (normally this is accomplished via CGI, but there are other methods). This means that the program handling the SOAP requests is started (or ran) every single time a request is received. The direct effect of this procedure is that the program looses its internal state in between different invocations.

While this behaviour is acceptable for some applications (for example a simple calculator, without memory), we need statefull behaviour. As it will be explained later (see section 2.3 on page 39), the server part of the application will need to continuously track the state of the simulated environment while answering to request from clients.

Despite the features provided by *Easysoap*, the library is still under heavy development. The latest stable version is 0.5 and the current development version is 0.6 (yet unreleased). All current versions of the library have a major problem, which has a huge impact on the performance of statefull applications. As the developers provide the source code for the library under the *GPL* license, I was able to track down the problem and fix it, enabling DALT to achieve an acceptable performance. The fix is described in appendix B on page 103.

2.1.1.4.2 XSoap The *XSoap*[7] package is used by all Java-based components of the project. The package is one of the few lightweight implementation of the SOAP protocol for Java. Most of the older Java SOAP implementations tend to grow into full-featured web application platforms. Using such an implementation would put an unneeded overhead, from both processing and configuration point of view.

[5] http://easysoap.sf.net
[6] http://abyss.sf.net
[7] http://www.extreme.indiana.edu/soap/

The XSoap package interfaces perfectly with the *Easysoap* implementation used by the C++ modules of DALT and it is fairly easy to use, even though there is no source of documentation except for the automatically generated API documentation which, unfortunately, is not complete.

For example, in order to use a SOAP method on the server which returns the dimensions of the map used by environment, the following steps have to be followed:

(i) First specify the fact that there exists a server which owns a method which returns the map size:

DALTServerService.java

```
1  import soaprmi.Remote;
2  import soaprmi.RemoteException;
3  public interface DALTServerService extends Remote
4  {
5     public int[] getMapSize() throws RemoteException;
6  }
```

DALTServerService.java

(ii) set up the SOAP client in the main code, by specifying the address of the server and setting a few attributes which direct the way in which the communication is performed:

Setting up the SOAP client

```
1  private DALTServerService serverRef;
2  private String location;///URL of the server
3  private XmlJavaMapping  mapping;
4  ...
5  try {
6      mapping = soaprmi.soap.Soap.getDefault().getMapping();
7      // disable SoapRMI auto mapping
8      mapping.setDefaultStructNsPrefix ("http://soapinterop.org/xsd");
9      // map SOAPStruct into namespace:http://soapinterop.org/ : SOAPStruct
10     mapping.mapStruct ("http://schemas.xmlsoap.org/soap/encoding/", "http://dalt",
11        "Entity", Entity.class,null, null, false, false, false);
12 } catch (XmlMapException x){
13     System.out.println (x.getMessage ());
14 }
15
16 try {
```

```
17      serverRef = (DALTServerService)
18      soaprmi.soaprpc.SoapServices.getDefault ().createStartpoint (
19           location, new Class[]{DALTServerService.class}, // remote service interface
20           "http://dalt", soaprmi.soap.SoapStyle.IBMSOAP,"http://soapinterop.org/xsd");
21  } catch (Exception e) {
22      System.out.println("Exception caught while setting up SOAP");
23  }
```

Setting up the SOAP client

(iii) once this is accomplished, the remote `getMapSize()` method can be called as an any other Java method. For example:

Using the SOAP client

```
1  int[] rez = serverRef.getMapSize();
2  if (rez.length != 2 ) {
3      System.out.println("This program only support 2-dimensional maps. Aborting...");
4      System.exit(0);
5  }
```

Using the SOAP client

2.1.1.4.3 SOAP.py This library is the easiest SOAP implementation to use. This is partially due to the library design, but mostly to the expressive power of the *Python* language. Similarly to *XSoap*, this library works perfectly with the *Easysoap++* library. The library was used initially to develop a prototype for the observer (see section 2.4.1 on page 63) and it later proved useful in the development process, for debugging.

In order to implement the same call to `getMapSize` all that is needed is the following:

SOAP.py example

```
1  #initialise the service
2  SOAP.Config.BuildWithNoType = 0
3  SOAP.Config.BuildWithNoNamespacePrefix = 0
4  server = SOAP.SOAPProxy("http://localhost:8000","http://dalt")
5
6  #get the map size,
7  map_size = server.getMapSize()
8  print "Map size is: ", map_size[0], ", ", map_size[1]
9
```

SOAP.py example

It is worth pointing out that *SOAP.py* does not require any definitions in order to specify the service capabilities, the way *XSoap* does (see point i on page 30).

2.1.1.5 Xerces

Xerces[8] is a portable XML parser, with compatible implementations for both C++ and Java (both of which are used by the project). The project uses the XML format to store data due to its flexibility. The task of debugging the output of different modules of DALT is made easier and the *ASCII* nature of the XML format enables the experimenter to modify some files by hand, which would be more difficult if the data would be in binary format.

Using XML for storage also gives platform and language independence. The XML files used by DALT are produced by a Java modules and they are read by both a Java module and a C++ one (see the map editor in section 2.4.2 on page 68).

The most important feature of the *Xerces* library (for this project) is its ability to parse and generate XML documents using the *SAX2* specification. The library comes with comprehensive documentation and a wide range of examples.

One of the features which differentiates *Xerces* from other similar libraries is its portabilities. The code for the library is highly portable and the developers report that the library is working on over 10 different platforms (ranging from Machintosh and Microsoft Windows to several flavours of Unix, AIX and OS/2).

2.1.1.6 Lyric

Lyric[9] is a C++ library which supplies a collection of functions, ranging from memory allocation and checked containers to time management. The feature which is most useful to DALT is its *Chronometer* class which can be used as a “real-life” chronometer (i.e. it can be started, paused and stopped in real-time). The definition of the chronometer can be as fine as microsecond or as coarse as seconds and minutes.

This library is important to DALT, as the simulation environment uses widely timing information in order to do runtime optimisations, benchmarking, etc. (see sections below for more explana-

[8] http://xml.apache.org
[9] http://lyric.sf.net

tions). Lyric is portable and isolates DALT from the machine-level details of working with the real time clock.

2.1.1.7 ZThread

As the C++ language does not have any threading primitives incorporated in the language an external library has to be used. The *ZThread*[10] library provides portable C++ threads for both POSIX platforms (most Unix environments) and Windows.

Being able to use threads is important for two reasons:

(i) the implementation of certain sections of the toolkit (see section 2.3.1.2 on page 40) is a lot clearer and more natural. This makes the initial development work easier and eases the process of implementing a simulation based on DALT;

(ii) by making use of several threads of execution the speed of the simulation can be greatly increased on multi-processor machines, as different threads can run in parallel on different processors. If the simulation would not be threaded the multiple processors would not be used to their full capacity.

While *ZThread* is one of the most advanced open-sourced C++ libraries, there are still some issues with it. While developing DALT repeated errors and seemingly random crashes prompted prolonged debugging sessions which revealed a very serious bug in all versions of *ZThread* up to (and including) version 1.5.4.

The problem lies within the locking mechanism used by semaphores, mutexes, guards and all other lockable objects. Under certain conditions (frequent thread creation/deletion, high number of threads, and many threads using the same lockable object) a lockable object such as a semaphore or a mutex can be acquired more than the specified number of times. This leads to several threads entering a mutual exclusion zone at once, which in turn leads to the corruption of the data which should have been protected by the mutual exclusion zone.

Once this problem was identified, it was reported to the Eric Cohen, the author of *ZThread*. He managed to reproduce the exact behaviour described and finally tracked down the problem to

[10] http://zthread.sf.net

the core of the locking mechanism used by the library and developed a fix for it. Now *ZThread-2.0.0a* (released on 24/02/2002) provides a reliable locking mechanism which enables DALT to run flawlessly.

Another issue worth mentioning is that Linux systems have an implicit limit for the number of threads that can run at the same time. The default limit (256 threads per process) is probably high enough for most applications but it is likely that a simulation which makes use of a high number of agents would need an increased limit. The procedure which needs to be followed in order to change this limit is detailed in appendix C on page 109.

2.1.1.8 Nbench

Nbench[11] is a portable benchmarking program, originally created by *Byte Magazine* and now placed in public domain. The program runs a series of computationally intensive algorithms, such as sorting, emulating floating point, compression, encryption and others. *NBench* measures the time taken by these algorithms in order to produce a set of results which describe the way the machine the test is running on compares to a reference machine (AMD K6 233 Mhz).

As shown in section 2.3 on page 39 it is very important to benchmark the machines taking part in the simulation as the benchmark results allow DALT to make certain predictions as to how different actions would alter the distribution of the load within the simulation system.

The program does not provide an unique figure to index the performance of the machine – it supplies different figures for the memory performance, floating point performance and integer performance. The authors argue that it is impossible to summarise the performance of the machine accurately by using one number. This is true if the benchmarking is intended for general use. However, it can be approximated what kind of computation dominates a given alife simulation. This can be used to create a weighed average of the results of the benchmark, which would actually summarise the performance of the machine fairly accurately with a single number.

In order to facilitate this, the source code for the benchmark has been modified in order to display the weighed average (with a default weight of 1 for all different figures). Code has also been added in order to read the weights from a file (`result_weights`, inside the `nbench` directory). Before running a long simulation, the weights should be modified to (roughly) reflect the

[11] http://www.tux.org/~mayer/linux/bmark.html

percentage of floating point, integer and operations within the computationally intensive zones. While this method is not going to provide a completely accurate indication of the performance of a given machine, it should provide an accurate enough approximation. This is supported by the results obtained using this method; see section 3.4 on page 82.

The modified version of the source code for *NBench* is provided within the source tree for the project, as each machine on which the simulation is ran will have to be benchmarked.

2.2 Toolkit overview

The top-level architecture of DALT is summarised in figure 2.1

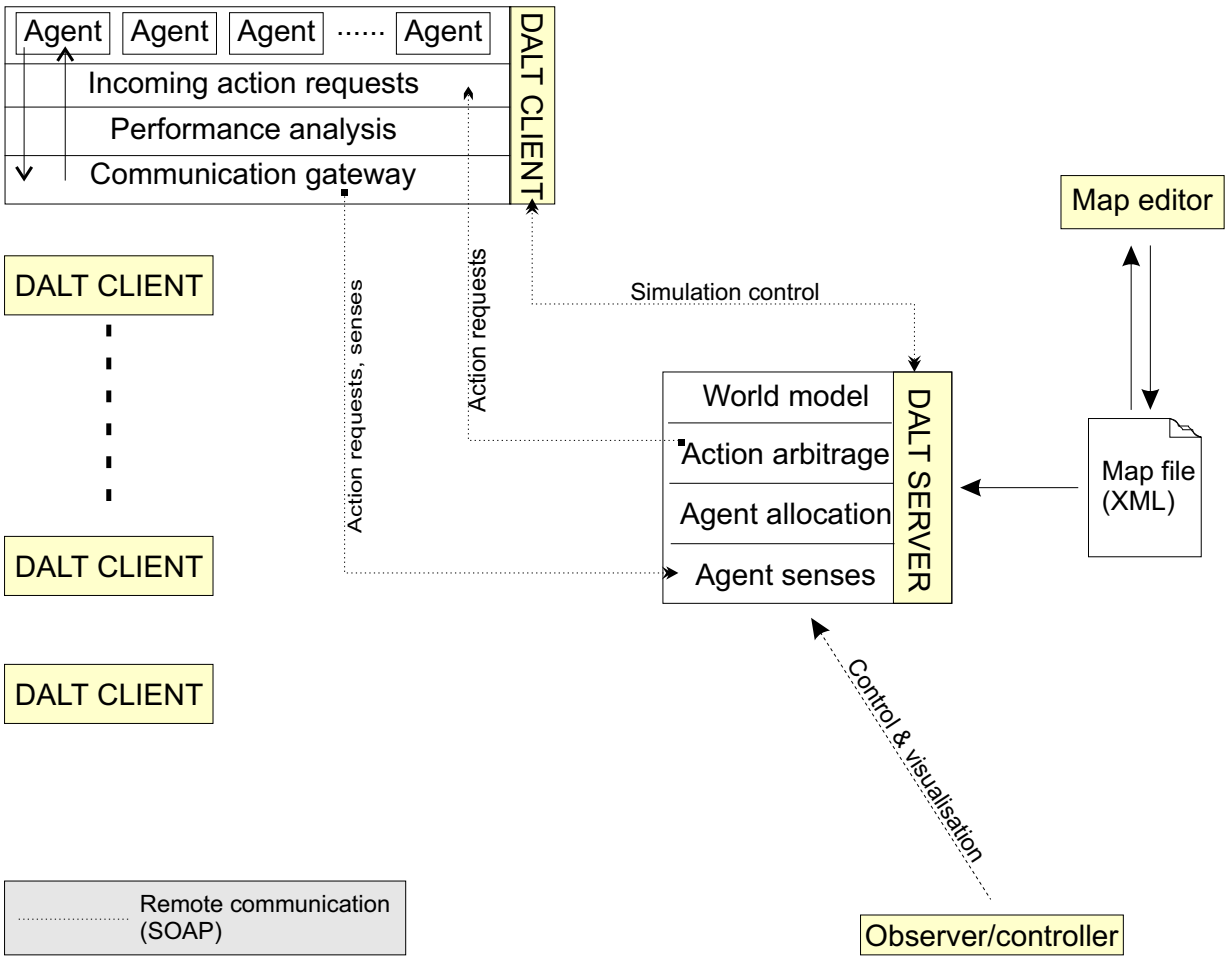

Figure 2.1: The DALT architecture

A simulation session usually starts with the creation of a map for the environment, using the *map editor*. Once a map is created the *server* can be started. As soon as the server is running

one or more *clients* can be started, on the same or different physical machines.

Finally the *observer* can be ran, at any physical location. The *observer* controls the running of the simulation by being able to instruct the server to run one or more processing cycle as well as providing a top-level view over the progress of simulation.

The different modules in DALT carry the following tasks:

the map editor is a tool for manipulating 2-dimensional maps for the simulation environments. The maps can be saved and loaded from XML files.

the server is a shared library which is used by simulations to create an abstract model of the simulated environment. The server drives the simulation (by manipulating its clients) handles agent allocation, arbitrates action conflicts and provides other functionality. Each simulation will only have one server.

the client is the processing unit of the toolkit. The client is a shared library which is used to create and run agents, the central part of the simulation. The number of clients in a simulation is only limited by the available network bandwidth.

the observer the observer provides a graphical view of the simulated environment. It also provides the means for controlling the running of the simulation (i.e. starting, stopping, pausing, stepping). The simulation cannot run without an observer.

communication while it is not a module per se, the communication protocol used in the interaction of different modules is very important for the simulation. A full reference of the communication protocol can be found in appendix A.

In the reminder of this section, a number of general issues and design decisions related to DALT will be detailed.

2.2.1 Distributing artificial life computation

When building a distributed application, the designer needs to choose very carefully what part of the computation can be parallelised, as this will directly influence the improvement in speed given

by carrying out the computation over several machines. *Amdahl's law* [Tanenbaum, 1999] states:

$$speedup = \frac{n}{1+(n-1)f} \tag{2.1}$$

where n is the number of processing units (in our case, DALT clients) and f is the fraction of the program which represents computation which cannot (or is not) parallelised. In the case of distributed applications (as opposed to parallel computation) the speedup will be further decreased by the slowdown incurred by communicating over a network, medium which is not as fast as the internal computer hardware.

In our case, f is the sum of all server tasks, which are:

1. loading and interpreting the initial map layout
2. (re)allocating clients
3. allocating agents
4. supplying agent perceptions
5. arbitrating and dispatching actions

Out of all these tasks, task 4 is the one which can potentially be most costly in terms of computation time. However this is completely depended onto the simulation which is being carried out. All other computations will be in $O(n^k)$, where the value of k is roughly equivalent to the dimensionality of the environment as their main task will be to interpret and filter the simulation environment in some way. This task can also put the heaviest burden on the communication infrastructure, by being a prime candidate for large data transfers.

Task 5 can potentially be a complex one, as there could be simulations in which a large number of actions is executed in every cycle. Depending on the simulation requirements, a complex algorithm might be employed to detect and resolve conflicting actions. However, is entirely dependent on particular simulation implementations.

The part of the computation which is parallelised by the toolkit is the agent processing. The only assumption the toolkit makes is that this part of the computation is actually significant. If this is not the case, the performance of the application running across several physical processing

units is going to be significantly inferior to the performance of the same application running on just one machine.

An exact value for the expected speedup cannot be extrapolated from this information. The speedup will vary greatly for different simulations due to their different nature and requirements.

An alternate architecture decision would be to construct self-sufficient programs, containing locally all the information needed to run the simulation for a given data set. The programs would only communicate when they need access to data managed by a different program. This would mean that each program would need to own a local copy of the world being simulated, which in turn requires every single program to be notified with every single change that occurs in the simulated world, or conversely, each program would need to check the rest of the community if its data is “dirty” before performing any operation. This communication overhead would be added to the normal inter-program communication. While approaches like caching, arranging the programs in a tree-like structure, etc. can be taken, the client would grow overly complex and other issues still remain open. Arbitrating resource conflicts would be extremely difficult. It would also be very difficult to ensure that the processing resources are used in an efficient manner.

Therefore, the choice was made to store the shared resources (i.e. the top-level information about the simulation world) in a single location, which is to be used by all other programs. Having a central location for world-level data can enable us to address all the issues mentioned above.

2.2.1.1 Network design for DALT simulations

The best suited architecture for the network supporting DALT is the star topology [Tanenbaum, 1999]. This model allows maximum throughput between the server, placed at the centre of the “star” and the clients. As the communication is carried over TCP/IP, the network layout is essentially irrelevant to the library itself, but usually most of the communication will be carried out bi-directionally in between the server and each client. The clients will never communicate directly with each other. It is important to optimise the layout of the physical network in order to accommodate this particular traffic requirement.

2.2.2 Simulation cycles

The toolkit revolves around the concept of simulation cycles. For an agent a simulation cycle contains last for as long as it takes the agent to execute one or more of the following:

- perceive the environment
- decide on one or more actions based on it perceptions
- react to environment stimuli

As shown in the next section, the simulation client contains one or more agents. For the client, a cycle last as long as the longest cycle of its agent plus the time required to perform the clients administrative duties.

A full simulation cycle (which is also the length of the server cycle) takes roughly as long as the longest client cycle. The simulation cycle is not precisely equal to the longest client cycle, due to the fact that the server might carry out extra processing before the start of the first client cycle, or after the end of the last client cycle.

2.3 The libraries

This section will describe the architecture and implementation of the two shared libraries which form the core part of DALT. This section is not going to give a detailed account of all the design and programming involved. The code is well documented and can be used together with the API documentation (available on the project website) for reference. Only the most important features will be highlighted.

C++ was chosen as the implementation language because of the features it provides and the speed of the compiled programs. The speed is extremely important for this project, as all the applications written using these shared libraries are going to be computationally intensive.

C++ code is also portable, provided the target platforms provide an ANSI C++ compliant compiler.

2.3.1 The client library

2.3.1.1 Overview

The client library servers as an “intelligent” container for the processing units of a simulation: the agents (see section 1.6.2.1 on page 21. As well as creating, running and destroying agents the client has to provide detailed timing information.

The client measures real time taken by each agent to compute, as well as the processor time[12] and the time taken by the client itself.

All timing information is measured at each running cycle (see section 2.3.1.2.1) and averaged over all the cycles performed. The timing information is further averaged by agent type. This information is very important for the server, as it is used to estimate the impact that the allocation of a new agent to a client would have over the client’s performance. The server continuously attempts to allocate agents in such a way that all the clients involved in the simulation finish each cycle in roughly the same time. This ensures that a new cycle can be started as soon as possible and no processing power is wasted by machines which finish quicker waiting for slower machines.

The client also provides a set of synchronisation features which shall be described later in this section.

2.3.1.2 Design

2.3.1.2.1 The agent cycle The agent architecture is roughly based on the one used in *Agent Orientated Programming* [Shoham, 1994]. Each agent is controlled by its own program module, and its core is based on a cycle.

The entire simulation revolves around the execution cycle performed by agents. It is very important to understand what is involved by this cycle.

In each cycle the agent can choose to examine its available sensory inputs. Based on what it perceived the agent can choose to attempt to carry out an action. Carrying out an action takes at least one cycle (some simulations may require certain actions to take longer).

An agent can also respond (in the same cycle) to an external action by either changing its internal state or carrying out an action.

[12] the time it would take for an agent to run **if** it would be the only program running using the processor

The part of the agent which carries out sensory processing and decides on the next action to perform is the most likely to require a high computational power.

2.3.1.2.2 Senses Each agent can have available any number of methods for perceiving its surroundings. Every single time the agent "uses" one of its sensory inputs, the agent will acquire information about a subset of its surrounding environment. In the case of DALT simulations this is most likely to be a number of other agents, as the all the elements composing the world can be described as agents.

The agent will only receive a subset of the *observable* information available. For any agent, not all its internal states will be known outside itself.

Definition 2 *We define an* entity *to be the highest abstraction of an agent. An* entity *is characterised by the following:*

(i) a set of coordinates, placing the entity within the environment

(ii) a unique identifier which is used to differentiate between entities

(iii) a type describing the entity

Agents have all the properties of an entity, as well as other properties which are dependent on the type of the agent. We shall describe an entity as being the *observable* part of an agent.

Therefore each time an agent will use its senses it will perceive a number of *entities*, or partial information about a number of entities.

This approach should be suitable to most simulations. However it is likely that other simulation would require the agents to perceive more information that it is available to the them through the entities. In these cases the experimenter is expected to enhance the definition of an entity by adding more pieces of information that should be observable.

As we shall see later, only the observable information about the simulation can be displayed and analysed by the *server* and the *observer*.

This approach is based on the blackboard model [Hayes-Roth, 1985], as all agents/actions are hidden unless made available to the server.

An important design decision has been made at this point: the senses will be (mostly) processed by the server. That is, depending on the sense being used by an agent, the server will attempt to reduce the amount which is supplied to the agent's sensory inputs by filtering out elements which are not likely to be processed. This process aims to reduce the amount of information which needs to be transferred between different physical locations.

2.3.1.2.3 Actions and action requests Most agents will require being able to execute actions such as moving, eating, picking up objects, attacking, etc.. The task of executing an action is split into two main parts: action request and action execution.

An agent can request an action by dispatching an action request to the server. The action request contains the name of the action (which uniquely identifies that action) together with other parameters. For example MOVE LEFT or KILL_AGENT 21.

The server will process the action request and if it is approved the action request will be forwarded to the target of the. In the example given above, the target for the MOVE action is the agent making the action request. The target for the KILL_AGENT action is the agent with the unique identifier 21.

Upon receiving an action request, an agent will execute the action corresponding to the action request received.

The agents are not the only elements of the simulation being able to request the execution of actions. The server itself may originate action requests on behalf of agents, or just as feedback to action requests previously submitted.

As we shall see in the library design the client library provides a generic class for defining action requests. An action requests is identified by the action name, the entity originating the action and the entity or entities which are targeted by the action. A constraint mechanism is also provided for enabling certain actions to be restricted to certain pairs formed by originator and target sets.

The restrictions are defined as a set of pairs. The first member of the pair is a set of entity types that can originate the action. The second member of the pair is a set of entity types which can serve as targets when the action is originated by any of the entity types from the first set.

2.3.1.2.4 Inter-agent communication DALT does not attempt to implement one of the many communication protocols/languages described in the specialist literature as different simulations will most likely adopt different approaches.

However, a common model can be used. The dialogue between two or more agents can be constructed on top of the existing action model. That is, each inter-agent communication can be modelled as a *communication action request*, in which the action requests carries a supplementary payload containing the message to be communicated.

2.3.1.2.5 Client library design The consolidated UML[13] class diagram is given in figure 2.2.

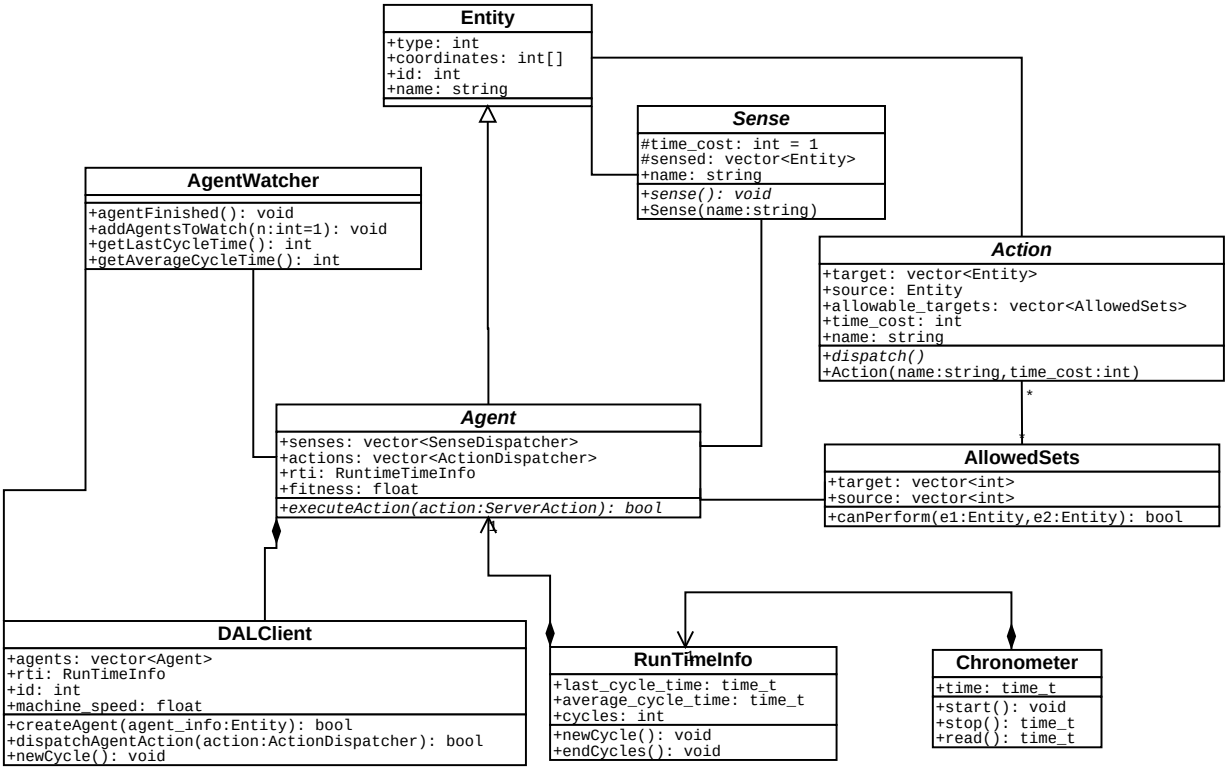

Figure 2.2: DALT Client UML class diagram

The classes have the following roles:

Entity provides the basic *observable* entity, as described in section 2.3.1.2.2 on page 41;

Sense provides the base definition for a senses, together with functionality for dispatching sense requests to the server. This class is purely virtual and cannot be used on its own;

[13] the notation in the UML diagrams used in this paper is the one standard one, described in Gomaa [2000], Fowler and Scott [2000]

Action provides the base definition for Action requests, as described in section 2.3.1.2.3 on page 42. This class does not provide any support for action execution. This class is purely virtual and cannot be used on its own;

RunTimeInfo provides support for tracking run time performance information. This includes both real-time and processor time figures, together with a mechanism for computing on-the-fly average run time values;

Agent gives a generic skeleton for an agent;

DALTClient is the main class of the client. It handles most of the communication functions, allocates and de-allocates agents and supplies meta-information to the server;

AgentWatcher tracks the time taken by the client to finish its cycle.

The way actions and senses are treated is slightly different from the way the toolkit was originally designed[14]. Also the need for the `AgentWatcher` class only became obvious during the implementations, for reasons that will be described in the next section.

2.3.1.3 Implementation and issues

2.3.1.3.1 Execution flow The main implementation issue is providing means for dealing with deadlocks and other synchronisation issues. Within the system we have a number of clients, all running in parallel and accessing (and being accessed by) the server. Each client has a number of agents, all running in parallel and accessing the server. The situation is made even more difficult by the fact that we need to avoid creating loops within the message paths.

For example, if some agent A_1 dispatches and action requests message M_1 targeting agent A_2, the server will eventually re-dispatch the message M_1 (provided that the action is approved) to the client managing A_2. A_2 cannot process the action request as soon as it arrives. If A_2 needs to check its sensory inputs, or if it needs to dispatch an action request in order to accomplish the action specified by M_1 it cannot do that, as the server is still waiting for M_1 to be handled and cannot receive any more messages until this is accomplished.

[14] the original design can be found on the project website: http://dalt.sf.net

Another example would be an agent which executes a MOVE action. This is accomplished, by requesting the action to be executed on itself as a target. As shown before, the message travels to the server for approval and then it is re-dispatched back to the agent. Now the server will need to move, but in order to do so it needs to notify the server of the change in position. Again, another deadlock, as the server is still waiting for the agent to finish moving. The message and execution flow for the client is summarised in figure 2.3. Another difficulty is ensuring the simulation flows

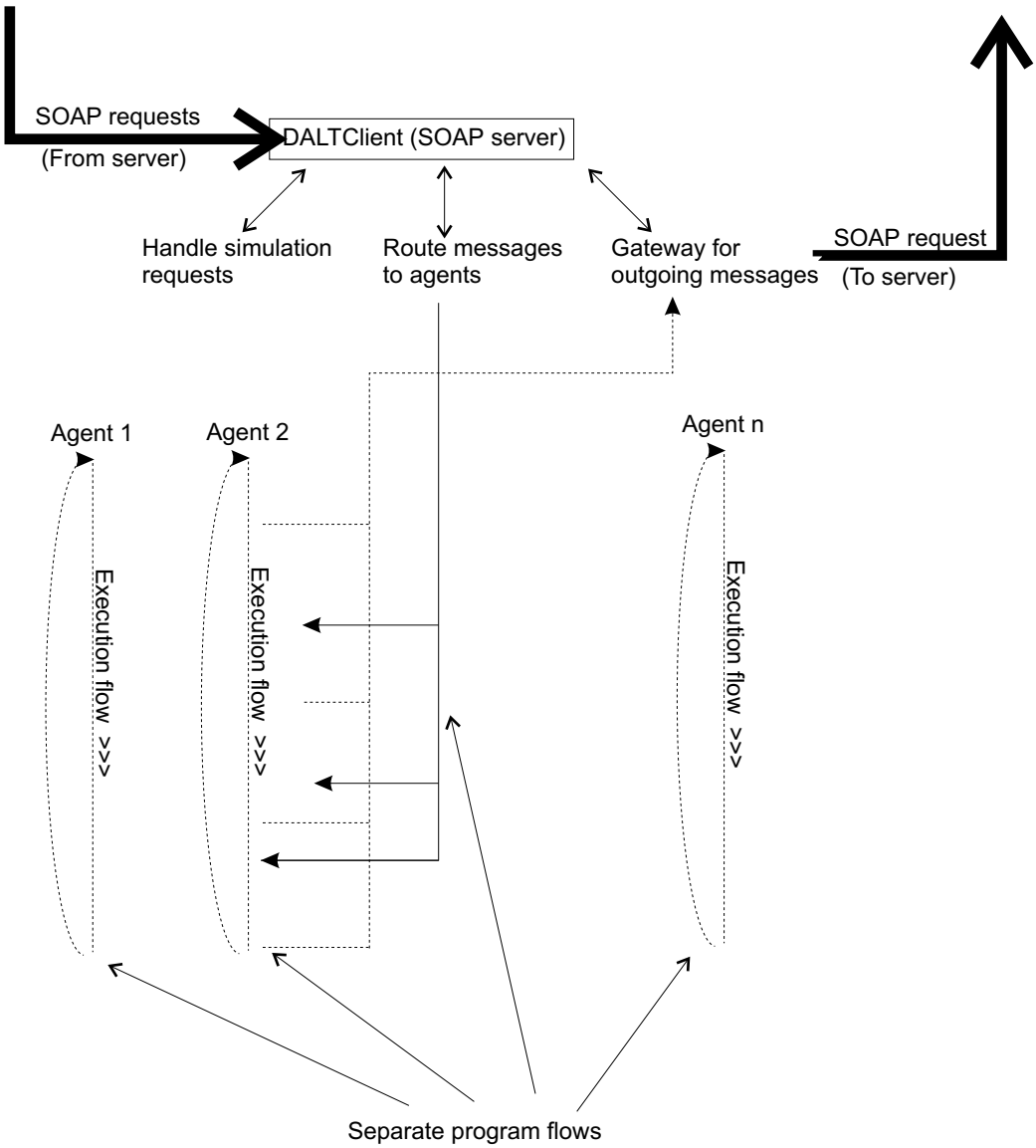

Figure 2.3: Message flow within the DALT client

correctly. In the case of the client, the main class, `DALTClient`, has no main function to execute. The class is idle until a message is received. Upon receipt one of the class methods will be called automatically. When the method reaches the end, the connection to the sender is closed and the

`DALTClient` object returns to the idle state.

While the `DALTClient` object follows this non-linear execution pattern, the agents managed by `DALTClient` run continuously, in parallel, sharing resources provided by `DALTClient`.

The most important shared resource is the SOAP gateway. As the toolkit attempts to split the low-level communication away from the agent implementation, both for reasons of performance and usability, the most efficient way for communication to be carried out was to place the SOAP gateway within `DALTClient`. This way all outgoing messages from the client will use the same object, which leads to increased speed and lower memory usage. However, as none of classes provided by the *Easysoap++*[15] library is thread safe, all the access to the SOAP gateway has to be placed in zones of mutual exclusion, to avoid two agents attempting to send messages at the same time. As the `DALTClient` object is executed non-linearly, it is not possible to run it in a separate thread, so none of the threading primitives (such as locking, mutual exclusion, etc.) can be applied to any code withing this object.

In order to address all the issues described above, a mechanism of delayed execution has been implemented. Each agent cycle has been split into a series of *sub-cycles* which follow the pattern described in figure 2.4. Within each sub-cycle, all the actions surrounded by square

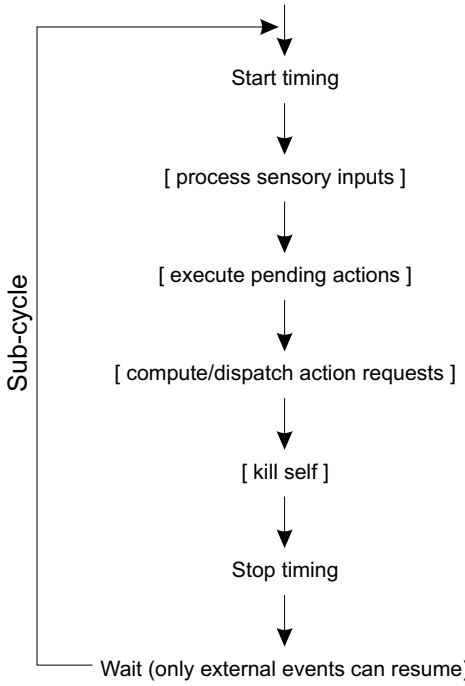

Figure 2.4: Subcycles within the agent cycle

[15] see section 2.1.1.4.1 on page 29

brackets may be performed 0 or 1 times. The execution runs through the sub-cycle until it reaches the end where the agent signals to the server that it finished its *cycle* and suspends itself.

However, external events might arrive during the execution of the sub-cycle. In the most simple case (see figure 2.5) the agent's execution pointer would be, for example, within the section processing sensory inputs when the `DALTClient` object would queue an action to be executed. Even though this is accomplished by calling the `queueAction` method within the agent, none of the code executed is thread safe due to the fact that it is executed from a function call of a non-thread object. However, the agent will not enter the next section until the `DALTClient` object finished queueing the action due to strict locking mechanisms preventing the agent from entering a code section which is currently being accessed from another program execution path. When the execution will finally reach the execution of pending actions, the actions queued can be handled gracefully. Messages can now be posted to the server as the `DALTClient` object ended the server's remote method call as soon as the action was queued. There can be another two

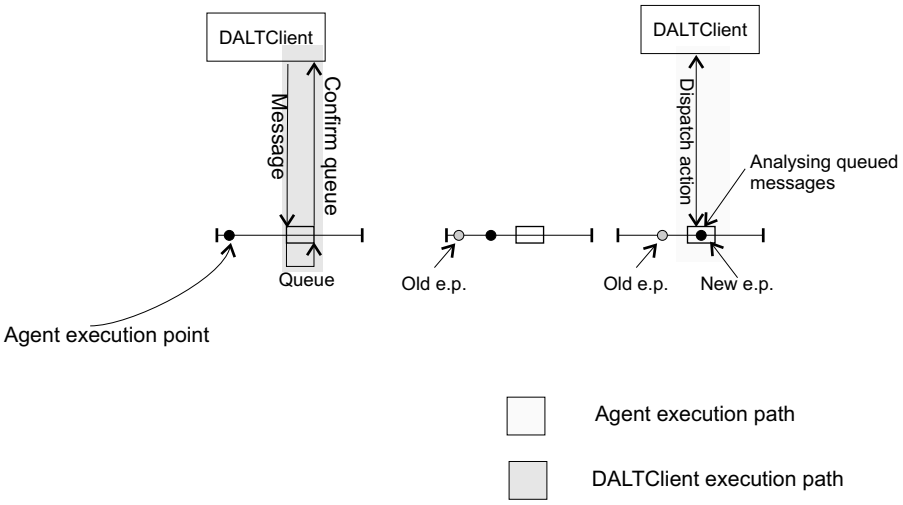

Figure 2.5: Responding to external events - case 1

more complex variants to this scenario as shown in figures 2.6 and 2.7 on the next page. They occur when the execution pointer within the agent program path has passed the point at which the execution pointer of the `DALTClient` object is trying to enter. For example, if the agent is deciding whether to kill itself while the `DALTClient` object queues one or more action requests. In this case the locking mechanism which causes the agent to wait at the end of each sub-cycle is disabled for the end of the current sub-cycle. The agent will continue by executing a new sub-cycle, in which

the only action it will perform is the one which has been affected by the `DALTClient`. In the case of our example, the agent would not check its senses, or try and kill itself. It would process its action queue and then wait.

It could easily happen that during the execution of the second sub-cycle a new message is received and it may be necessary to move to the third sub-cycle, and so on. Whenever the

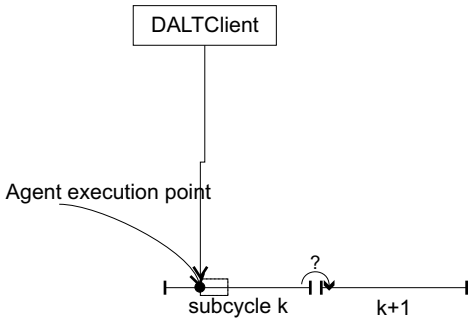

Figure 2.6: Responding to external events - case 2

execution pointer within the agent is right at the start of the zone that is about to be entered by the `DALTClient` execution path, the locking mechanisms will non-deterministically reduce the conflict to one of the two scenarios described above, by letting one and only one of the execution paths carry on.

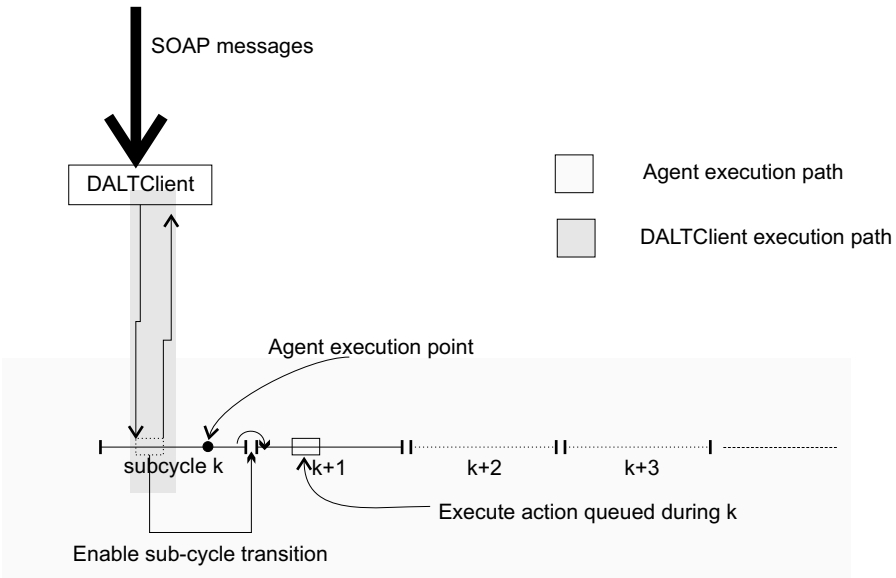

Figure 2.7: Responding to external events - case 3

2.3.1.3.2 Measuring time The pattern of execution described above raises some interesting issues when trying to measure the execution time for a given agent. If the **real** execution time for an agent is obtained, this time can be combined with the information about the machine speed obtained from benchmarking to estimate how long it would take for the same agent to be executed on a different machine. Unfortunately, the way most `Chronometer` classes work is by reading the real-time clock on start and stop, and returning the difference between the two times as the time elapsed.

While this approach is reasonable for a machine which is running just one agent, as soon as several agents run at the same time, the time taken to run an agent will increase proportionally to the number of agents running on the same machine. Simply dividing the real-time taken by the number of agents that are running would give a wrong result, as some agents might have been running for longer then others. It is also possible that some of the agents received a greater proportion of the time-slices allocated by the processor.

However, as each agent runs in a separate process we can obtain from the CPU the time that has been spent executing only that process as well as the time spent executing system calls on behalf of that process. By manipulating these times in a similar fashion to the one used for timing single-threaded programs we can obtain accurate timing information for each agent. This functionality has been incorporated within the `RunTimeInfo` class.

Another difficulty in measuring time is given by the need to measure the real-time taken by each client cycle. This is the time interval measured from the moment when the client instructs its agents to start a new cycle, until each agent finished its cycle. The difficulty lies in the following:

(i) the agents themselves do not know when a cycle is finished. The agents perceive the execution flow as a sequence of sub-cycles, at the end of each sub-cycle the chronometer is stopped and the server is notified that they ended the cycle. However, external actions may trigger the agents in performing a new sub-cycle.

(ii) the `DALTClient` object has no way of determining whether all the agents have finished processing

In order to address these problems, the `AgentWatcher` class has been created. This is an object running on a separate thread of execution, that starts its own chronometer (if it is not

running already) whenever a message is received by the client. This object also keeps track of how many agents started a sub-cycle and how many finished their sub-cycle. The `AgentWatcher` will keep the chronometer running for as long as there are agents still within their sub-cycle.

On the start of a new cycle, the chronometer is reset and the timing for a new cycle starts. The time information for the old cycle is backed up so that whenever the client is enquired about its time performance it can return valid information.

2.3.1.3.3 Agent destruction The final major implementation issue is the destruction of agents. The problem resides in the fact that in order to destroy an agent multiple actions have to be performed:

- the agent has terminate all its activity cleanly, without disrupting the simulation;
- the agent cannot maintain exclusive access to any mutual exclusion zones;
- the agent has to exit its sub-cycle loop;
- the `DALTClient` object has to dispose of all the resources allocated for the agent cleanly. This includes both the agent object itself and all other statistical or reference data available about that agent;
- in order to terminate the task within the operating system, the agent thread has to be joined[16] within the execution of the `DALTClient` object. If this step is ignored, dead threads un-disposed will quickly accumulate, preventing the allocation of new threads.

Most of these actions cannot be performed by the agent itself, for reasons which shall be described below.

In order to handle these cases, another delayed mechanism has been implemented. The `DALTClient` object intercepts all the KILL messages before notifying the agent that it should terminate. Each agent handles its internal clean-up. All the processing that is being carried out

[16] consider the following situation: there are two programs P_1 and P_2 running in parallel. The program P_1 has the statements $s_1, join(P_2), s_2$ and program P_2 has the statements $k_1 ... k_n$. The `join()` statement will result in the execution of all the statement from the current execution point in P_2, say k_i to the last statement, k_n. Once the last statement in P_2 is executed the execution of P_1 is resumed with s_2

by the agent at the moment at which the KILL message has been received is finished. The agent then finishes its current cycle and notifies the server of its "death". Meanwhile the `DALTClient` object will queue the agent for resource clean-up.

The queue of agents to be cleaned up is processes at each simulation cycle. All the agents that exited their sub-cycle loops are joined. Joining the agent threads has as only effect the cancellation of the thread supporting the agents, as their execution has already ended. Finally all internal simulation data referring to the destroyed agents is disposed.

2.3.2 The server library

2.3.2.1 Overview

The server library supplies the following functionality:

- being able to load the initial state of the simulation together with characteristics of the environment from a file;
- allowing clients (programs using the client library) to connect at any point during the progress of the simulation;
- resource management;
- supplying agents with the sensory inputs;
- arbitrating resource conflicts for actions requested by agents;
- allowing particular implementations to use the server to run a custom routine which processes the simulation environment. This can be very useful in applications using genetic or evolutionary algorithms;
- providing a centralised simulation model, containing all the observable simulation details.

Not all this functionality is fully implemented by the toolkit within the server. For example the server provides the capacity for arbitrating actions. This is accomplished by queueing all actions request without actually dispatching any until all agents finished sending action requests. Then

the experimenter has the option of processing this action queue into approved and rejected actions. The approved actions will be executed and the rejected ones can be processed further if necessary. The library does not provide an automatic mechanism for identifying conflicting action requests. Doing so would limit the scope of the application by needing to specify strict formats for action requests and what constitutes a “conflict”.

The same is true for a few other functions which are highly dependent on individual implementations for different simulations. As it will be shown later on, in these cases the toolkit provides most of the base functionality, leaving up to particular simulations implementations to build on top of the model provided.

2.3.2.2 Design

The server has a rather unconventional architecture, as it does not have its own thread of execution. All the actions performed by the server are performed as a direct reaction to incoming messages.

The observer (see section 2.4.1 on page 63) provides the messages which enable the server to drive the simulation. While this may seem as an unimportant implementation detail, it has a profound effect on the way the server is designed.

The observer triggers the server into loading the initial environment layout by sending its first message. Subsequently, the observer will continuously send messages to the server at set time intervals (usually around 600 ms), checking whether the current simulation cycle has ended.

The server uses code triggered by the incoming observer message to detect when a cycle has ended and to implement some other functionality.

2.3.2.2.1 The environment The environment is modelled as an n-dimensional space. As most simulations will only make use of a 2-dimensional space, all the graphical tools supporting the toolkit will only work with 2-dimensional environments. Both the server and the client support n-dimensional sets of coordinates for all operations.

The parameters defining the environment, such as its dimensionality, the span on each axis and pre-existing agents, are read on start up from a XML file. The format of the file is described in section 2.4.2 on page 68.

The server has an internal model of all the observable[17] agent details. This model is continuously updated as the simulation progresses.

2.3.2.2.2 Clients, agents, performance tracking Once the server is running, clients can connect to it. As soon as a client is acknowledged the server retrieves all information about that client, such as the address at which it can be contacted and the benchmarked performance of the machine it is running on.

Usually clients connect to the server before the simulation is started. This enables the server to distribute the initial agents evenly between all available clients. This can be very important if the simulation will only use a limited number of agents, without creating new ones frequently. However, the server allows for client to log in dynamically after the simulation has been started. The newly logged in clients will immediately become candidates in the agent allocation process.

As the simulation progresses, the server keeps track of the average time taken by the client to compute its computation cycle, together with the time it took to compute its last cycle. The server also keeps records of what agents are running on each client.

The server gathers statistical data from all clients, reporting the average normalised[18] time taken to compute each agent type. The statistical information is averaged to produce a list of normalised times for each agent type in the simulation. This approach should compensate for the fact that benchmarking is imprecise.

2.3.2.2.3 Resource managment and agent creation The statistical information about the clients and available agent types is used by the server whenever a new agent is created. In order to create a new agent, the simulation environment has to be updated, the an agent object has to be physically created within one of the clients connected to the server.

In order for the simulation to run efficiently, the server attempts to keep the time gap in between the first client which finishes its cycle and the last client which finishes its cycle to a minimum, as a new simulation cycle cannot be started unless every single client finished processing.

Therefore the server has to make an informed choices about the client on which a new agent

[17] see section 2.3.1.2.2 on page 41

[18] i.e. the time it would take to compute the agent on a reference machine, based on the performance index given by the benchmarks ran on the machines holding the clients

is to be allocated, and the impact that this allocation would have both on the time taken by the client to finish its processing cycle and the way this would affect the entire simulation.

Each time a new agent has to be allocated to a client the server acts as a dynamic (on-line) scheduler [Kopetz, 1994]. Each client is viewed as an execution task and each agent as a block of instructions. The scheduling is entirely centralised. The interesting problem raised by this situation is that the scheduler cannot change the priorities of the tasks directly. It can do so by adding/removing processing load from the tasks. In a standard setting it is quite likely that the "tasks" (i.e. clients) will have different priorities. That is, the physical machines they are the clients are running on are likely to have different speeds. The goal is to allocate work units in such a way that all tasks finish at the same time. The rationale behind this is that a computing cycle is as fast as the slowest client taking part in the simulation.

s bit There are three allocation modes that can be used in DALT:

random in this mode, an random client is chosen for each new agent;

biased random in this mode the clients are still chosen at random. However, each client's probability of being chosen is directly proportional to the processing power of the client;

greedy allocation in this mode follows the following algorithm:

- for each client c_i, the performance impact that allocating the agent on that client would have over the client performance is calculated. This is accomplished by studying the way that the client performance would modify in relation to the average time taken by a client to finish its cycle. The simulation would be making the most efficient use of resources when all the client cycles finish at the same time. For most cases, it is safe to assume that at each cycle the time the clients take to complete their own cycle should tend to is the average of all client cycle times. The improvement for each client is calculated by measuring the way the distance between the current time and the predicted time, should an agent be allocated.
- allocate the agent on the client which shows the best improvement.
- if there exist two or more clients sharing the same "best improvement" the algorithm chooses one of them using the *biased random* method.

The greedy algorithm above attempts to find the best client at each run. However, this is only a local optimum. It is virtually impossible to develop an algorithm which would have an optimum global performance as there are too many factors which are unknown and can only be determined (if they can be determined at all) for particular simulation implementations. The server has no way of foretelling how many agents will be allocated during next cycle (if any) and how this number would change in future cycles. The server has also no way of foretelling what agent will be destroyed, thus altering the performance of the clients that host them. Another important factor is the number that describes the performance of each machine. While this number is a good indication of the performance it is by no means accurate (see section 2.1.1.8 on page 34).

Yet another element of uncertainty is introduced by the fact that it is entirely possible that agents might not have similar execution times. It could be the case for some simulations that the agents of a given type will take a completely random amount of time to execute at each cycle. The server will do its best to average all these times in order to provide an approximation for that agent type, but this may have a significant impact on the performance of the simulation.

The best approach in this situation is to allocate each agent while trying to obtain a local optimum. This provides (see section 3.4 on page 82) a good enough performance, without using too much processing power.

For some applications it might be desirable for agents to move from one client to another. The main reason for trying to accomplish this would be to handle the case in which the simulation is running for several cycles without any change within the agent population (births and deaths) and the clients are unbalanced in terms of the length of their respective cycles. In such a situation it may be necessary to move an agent from a client to the another one. This can be accomplished in the following way:

- the server chooses an agent which is *clonable*;
- the server sends a CLONE action request to this agent;
- the agent sends a message to the server containing a request for creating a new agent, together with enough internal data for another client to be able to duplicate the original agent;

- when the agent is created the server sends a KILL message to the original agent.

The agent to be cloned can be chosen by employing another greedy algorithm. For each agent type, the server can attempt to remove one agent of this type from each client and then attempt to re-allocate the removed agent on each client, calculating the improvement at each step. This procedure can be repeated until no agent movement which might improve the simulation time is found.

This algorithm has a relatively high complexity. The complexity is in $O(n^3)$ if the procedure is ran only once, or in $O(n^4)$ if it is ran several times. However, the value of n is small. Most simulations, even ones with thousands of agents, only have a small number of agent types. Also the number of processing clients is very unlikely to exceed one hundred.

An even more radical approach would be to remove all agents from the simulation and re-allocate them as if the simulation would be started at that point. In this case only the agents which would get re-allocated to a different client would be physically transferred from one client to another.

The main issue with the cloning system is the time taken for transferring agents. As the transfer takes place over a slow[19] network, the simulation can be negatively affected by this procedure.

Each experiment has to implement its own serialisation procedures for clonable agents. There are dedicated serialisation libraries that are able serialise automatically any object. However, using one of this libraries would serialise a lot of unnecessary data (such as code for methods, data structures that are not needed for reconstituting the agent). Transferring this data is superfluous and would affect negatively the available bandwidth and therefore the overall speed of the simulation.

2.3.2.2.4 Resource arbitration A big issue for most simulation is the way resource conflicts are solved. Some situations can simply be solved at agent level. For example consider a simulation which involves rabbits and big rocks with the constraint that a rabbit cannot occupy the same place as a rock, or other rabbit. One of the most intuitive examples of conflicts is a rabbit trying to move on a location occupied by a rock. This situation can be easily solved by the rabbit agent by

[19] compared to the speed of internal memory transfers within a computer

checking its senses (assuming that the rabbit has eyes and it is not blind) and not attempting to move to locations containing rocks.

However, it could be the case that the rabbit cannot use its eyes and decides to move on to the location with the rock. This type of conflict cannot be solved by the rabbit agent itself, as it does not know that there is a rock at the location where its trying to move. The conflict cannot be solved by the rock agent either, as it would not be able to perceive that a rabbit is trying to displace it.

This type of situations lead to the creation of a mechanism for arbitrating resource conflicts. The arbitration is based on the assumption that at any given cycle the state of the observable environment is free of conflicts. The only way conflicts could arise is as a consequence to actions performed by agents. This is one of the main reasons for which all action requests are sent through the server. The server does not forward any requests to their targets as they arrive. The action requests are instead queued.

When all the agents finished posting their action requests and the server had queued its own action requests (if any) the action request queue can be processed. The server splits the action requests queue into two sets: a set of action requests which are approved and a set of action requests which generate conflicts. The approved action requests are forwarded to their targets. The conflicting action requests can be treated in a number of different ways:

- they can be all denied and subsequently discarded;
- some of them can be approved (based, for example, on the fitness of each agent) and the rest can be discarded;
- all action requests can be returned to the agents originating them, allowing the agents to either deal with the refusal of the action request internally or pick an alternative action request.

The implicit toolkit behaviour is to queue all the actions and call a set of methods which carry out the sorting of action requests into subsets. Other methods can be used in order to deal with the resulting subsets of action request. Each simulation can customise the way in which the actions are split to subsets and the way conflicting action requests are handled.

2.3.2.2.5 Providing simulation data The artificial life experiments are ran in order to collect data which is to be interpreted by the experimenters. All the data about the observable environment can be logged into one or more files to be processed later with specialised graphing and/or statistical tools, such as *R*.

However, some simulations will need to display the state of the environment graphically, in real time. In order to achieve this the server provides functions that can be accessed remotely which return generic information about the environment, the entire state of the environment or the set of changes from the last state of the environment.

2.3.2.2.6 Sever library design The consolidated UML class diagram for the DALT server is given in figure 2.8 on the next page.

The classes have the following roles:

Action provides the representation for an *action request*. This is the representation used by all other classes when dealing with action requests.

ActionArbitrage provides all the functionality for arbitrating resource conflicts (see discussion in section 2.3.2.2.4 on page 56).

Client provides the representation of a client, including its location, performance, its current processing status and a method for creating agents on this client.

ClientManager holds all the information about agent locations, available clients and the performance of agents and clients. This information is used to implement the various agent allocation and management strategies described earlier.

WorldMap provides an abstraction for the environment, allowing each location in an n-dimensional environment to be accessed independently, by its coordinates. The class also implements functions for tracking the changes in the environment which occur during each cycle and allows for environments to be loaded from external sources.

DALTServer is the main class of the server library, linking the functionality of all other classes. This class also provides the entry point for all the methods which can be executed remotely by the clients and the observer.

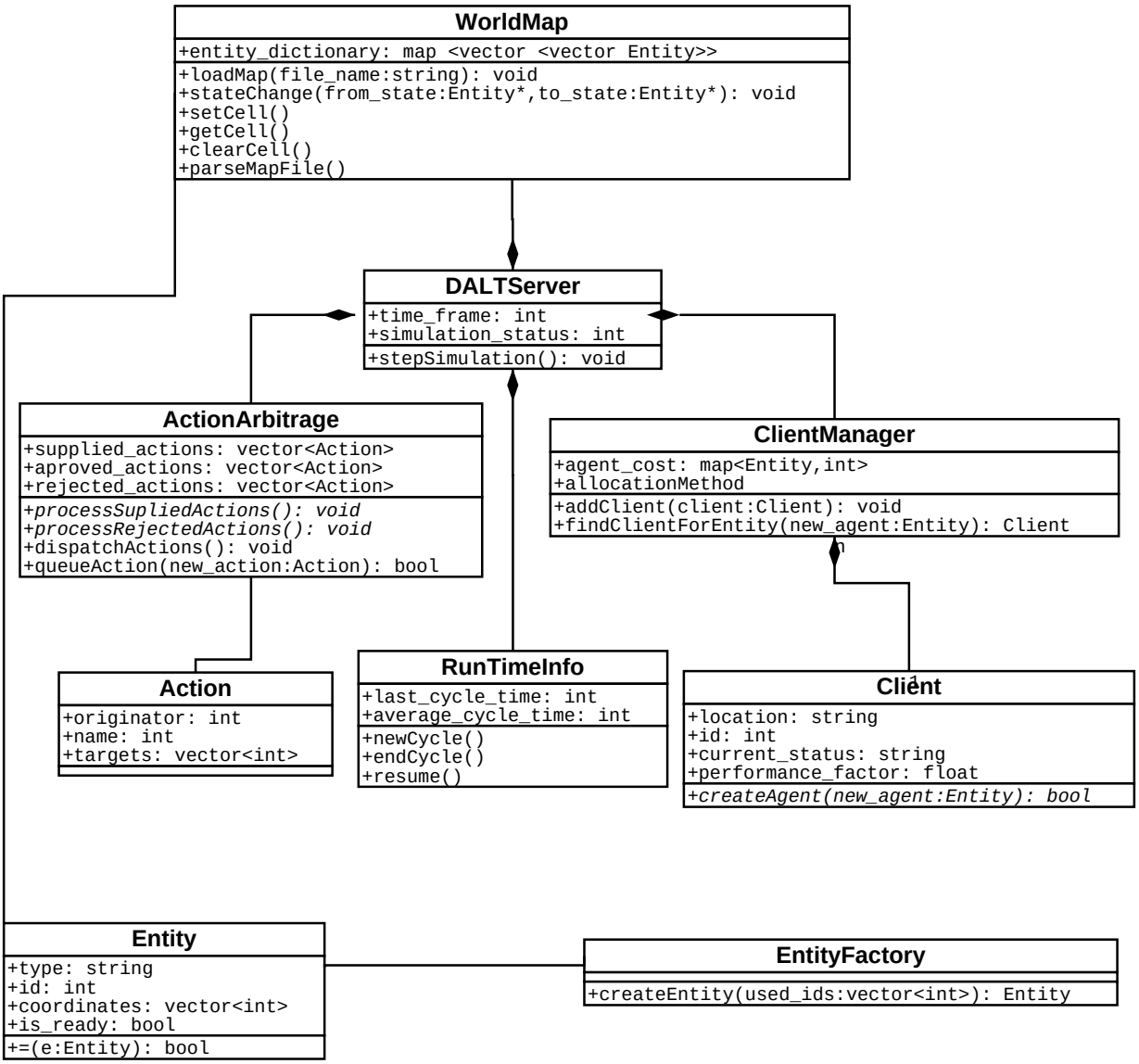

Figure 2.8: DALT Server UML class diagram

RunTimeInfo implements a chronometer which is used to measure the duration of each server cycle.

Entity represents the observable model of an entity. It is roughly equivalent to the model described in section 2.3.1.2.2 on page 41.

EntityFactory is used to create new entities. The main role of this class is to ensure that all existing entities have an unique identification number.

2.3.2.3 Implementation and issues

The implementation of the server faced less technical problems than the client implementation. Most of the issues were addressed in the design stage.

As the server is entirely driven by external messages, it does not have its own execution thread. All the external messages are accepted by the SOAP server, serialised, queued and passed one by one to the methods in the server designated to handle them. The periodic polling messages received from the *observer* are used to perform periodic diagnostics on the internal state. This approach has been chosen in order to minimise the complexity of the observer (this topic will be expanded in section 2.4.1 on page 63).

2.3.2.3.1 Entities The entities get allocated an unique identification number (UID) by the `EntityFactory`. The `EntityFactory` class is unique and globally visible within the `DALTServer` namespace. Each time the `DALTClient` object needs to allocate a new `Entity` it will use the `EntityFactory` object to allocate the new `Entity` and assign it a valid UID.

UIDs are allocated starting from 1. UID 0 is assigned to the server, whenever it serves as target or originator for an action request. The UIDs are allocated sequentially, using a 32 bits unsigned integer. In the unlikely event that the current UID reaches the maximum, the allocation procedure will loop the UIDs, trying to allocate UIDs starting with 1 again.

This is done in the hope that some or most of the agents allocated initially have been destroyed leaving their UIDs available for allocation. Once the allocation mechanism loops, at each allocation the proposed UID has to be checked against all the currently operating agents UIDs as each UID is likely to be assigned to an active agent.

2.3.2.3.2 Server cycle Each server cycle starts with a NEW_CYCLE message from the observer. The server notifies each known client of the event. The clients in turn notify all their agents of the start of the cycle. As well as notifying the clients, the server will retrieve the updated performance information for the previous cycle. Once all the clients are notified the server will return to its idle state waiting for all the agents in the simulation to announce themselves ready for a new cycle.

As the server cannot continuously check through its own means whether all the agents fin-

ished processing. It relies on the observer instead, to periodically enquire whether the system is ready for a new cycle. The server will then check whether the agents are ready in response to the requests received from the observer. If the agents are ready, the server will run its own computation on the environment (if any) and it will then check whether there are any queued action requests.

If there are no queued action requests and none of the agents re-started their computation cycle due to an action taken by the server within its computation cycle, the server will close the current cycle, notify the observer about this decision during the next observer communication. The server will then transfer itself to the idle state again, waiting for other messages.

If there are action requests within the server's action queue, the requests are processed and the valid ones are dispatched to the relevant agents. All the agents affected by these actions will start a new sub-cycle and the server will repeat the whole process again, waiting for agents to finish and then processing action requests.

Particular simulations can instruct the library to execute the processing cycle of the server only once per simulation cycle (the default behaviour) or each time the action queue is processed. This feature can be extended easily to enable simulation to execute the server processing cycle at set cycle intervals.

The entire process described in this section is implemented by the `DALTServer` class.

2.3.2.3.3 Using n-dimensional environments The initial state of simulation is stored in an XML file (see section 2.4.2 on page 68) which is parsed using the *Xerces* library. The entire environment is stored within a vector.

The contents of the this vector can be accessed using access methods taking as parameters variable length vectors containing a list of coordinates. This method allows for the same library to be able to operate correctly with environment with any number of dimensions, without using any the needing to change the code or to use the "old-style" C functions with variable number of argument. Each location in the environment can contain no more than one agent.

A planned extension which would allow more then one agent at each location has been considered but not fully implemented. The environment would store a list of agents for each one of its

locations, rather then just one agent. This would allow for modelling mostly flat worlds[20], without the memory over-load of increasing the dimensionality of the environment.

The `WorldMap` class can use two sources for the representation of the environment. One of them is the vector which fully describes the environment. However, for large sparse environments this could require allocating unnecessarily much memory. A second representation is provided: all the agents, together with complete information about all their observable characteristics is stored separately. This avoids the memory overhead of allocating memory for locations not containing agents. For some simulations it might be desirable to use this source of information rather then the default representation. It is worth pointing out that managing senses using the second representation can be more difficult and computationally inefficient, as the entire list of agents would have to be parsed just in order to figure out a seemingly simple thing, like obtaining a list of neighbours for a given agent.

It is up to the designers of individual experiments to decide which method is more suitable for their application. The toolkit uses both methods by default, continuously updating both representations.

2.3.2.3.4 Observable events The `DALTServer` handles all the incoming reports about agents changing their observable characteristics. In its default configuration the only observable attributes are the ones described in section 2.3.1.2.2 on page 41. All the changes in the observable environment are incorporated in *state_change* reports by the agents.

The creation of a new agent, for example is described by a transition from an empty state to a state containing a valid agent, with a type, unique identifier and location in the environment. Similarly, the destruction of an agent is represented by a transition from a valid, existing agent to an empty state. Movements can be described as transitions between similar observable states in which the position is the only changing attribute.

Of course, the list can be extended to include other custom attributes, such as energy, or age. These attributes would have to be added both to the basic entity described in section 2.3.1.2.2

[20] for example 2-dimensional world with rabbits and patches of grass where the rabbit can move on top of a patch of grass; this could be modelled with the current system by creating a 3-dimensional environment with the z axis allowing for only 2 different heights but would cause the server to allocate a large amount of memory

and to the basic entity provided by the server.

All transitions which occur in a simulation cycle are used to update both models of the environment **and** they are queued. The transition queue is emptied whenever the observer solicits a list of changes. For most simulations, transmitting only the transitions occurred from one cycle to another can be less expensive in terms of networking bandwidth than transmitting the entire state of the environment at each step.

However, as both methods are provided, the experimenters can choose one or the other depending on the requirements of the simulation.

2.4 The tools

This section gives design and implementation details about the additional tools provided by DALT.

2.4.1 The observer

The observer drives the simulation by controlling the server directly. The control is two fold. The observer can instruct the server to execute a simulation cycle and therefore it can use this to provide features like running the simulation continuously, pausing, resuming and running the simulation step by step.

In order to be able to instruct the server to run a new simulation step, the observer needs to know when the server finished its current cycle and returned to its idle state. This is accomplished by continuously “asking” the server whether it is idle and only launching a new cycle when the server is idle. The server would simply refuse to execute a new cycle when a cycle is currently running. The continuous polling is used by the server as a trigger for managing some of its own activities and is required for the simulation to run.

For a full list of the messages than can be exchanged in between the server and the observer refer to appendix A.

2.4.1.1 Design

The conceptual model for the observer is shown in figure 2.9 on the next page. At the core of the server lies the polling mechanism described in the previous section. This module also gathers

data from the server. The data obtained this way is interpreted in a separate module, which

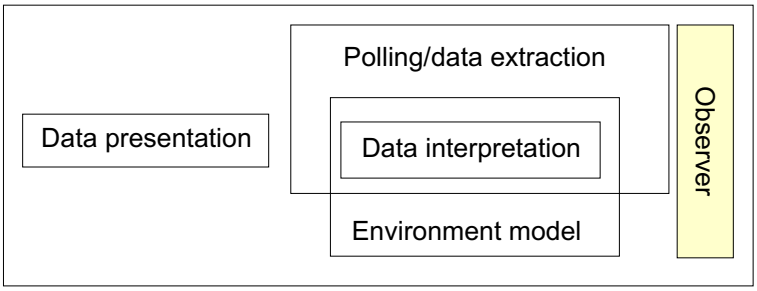

Figure 2.9: Conceptual model for the observer

also updates the internal model of the environment. The environment modelled by the observer is not necessarily the same as the one model by the server. The observer has access to all agent information available to the server, but it is quite possible that the observer would interpret this data more thoroughly. For example the observer might track inter-agent relations (such as social structure, or an agent's affinity for another) and would need to model this internally before displaying it graphically. The server would not necessarily have to model these relations.

A separate module deals with presenting the data to the experimenter, by displaying it graphically and/or logging it.

A UML diagram of a observer being able to display the agent layout in a 2-dimensional environment is given in figure 2.10 on the facing page.

2.4.1.2 Implementation

Initially the design specified that the server would notify the observer about changes in the observable environment. However, it became obvious during the implementation that this approach would increase the complexity of the observer unnecessarily. The initial approach would also have complicated the server by requiring the server to adopt a threaded model similar to the current client model.

The current approach is much more efficient in terms of code complexity and reliability at the expense of a slightly less intuitive model.

2.4.1.2.1 Basic observer A basic observer can be implemented in about 30 lines (using the Python programming language). The observer simply runs the simulation endlessly, without offer-

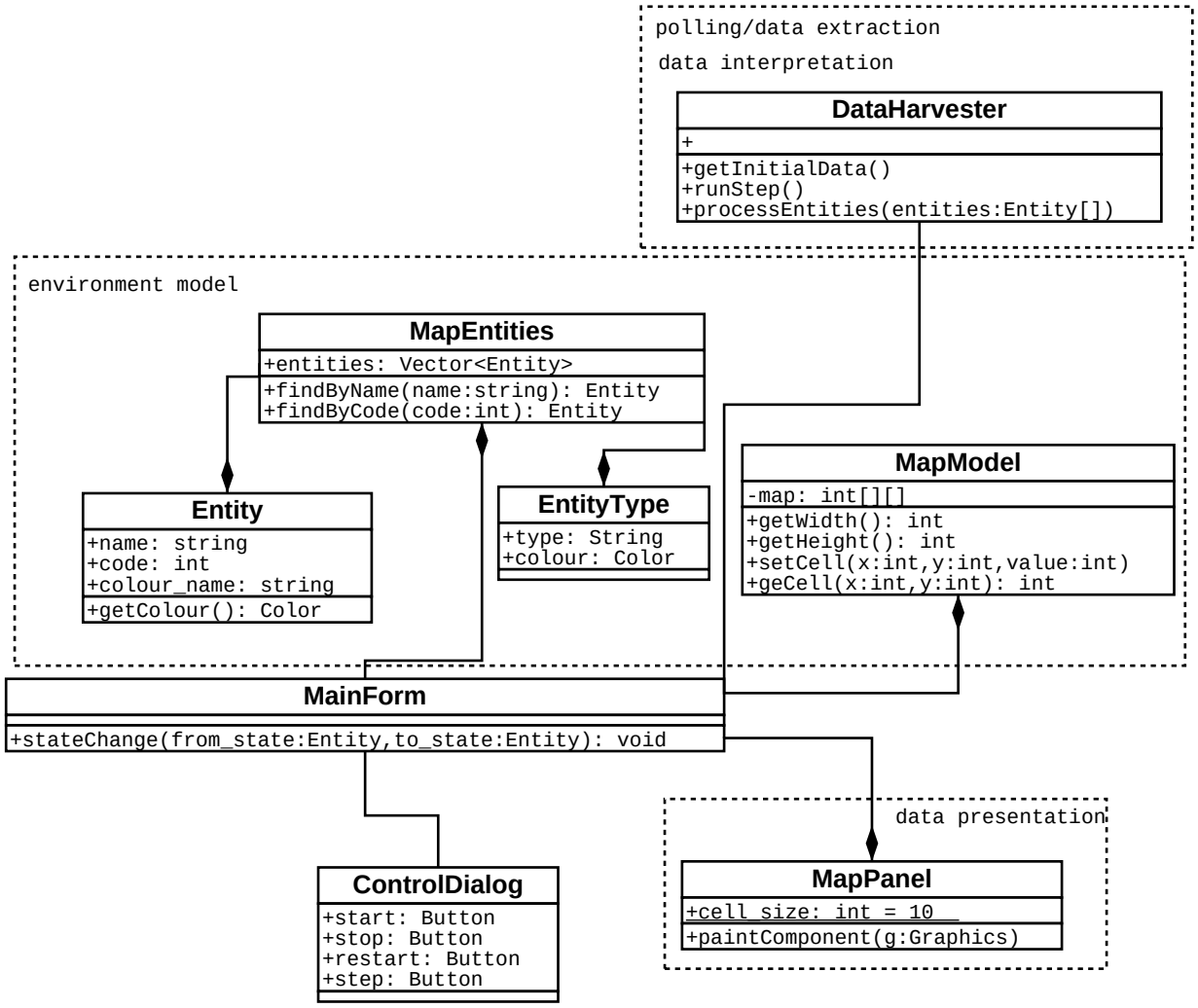

Figure 2.10: UML model for 2-d observer

ing any control over it, or any feedback. This fully implements the *polling/data extraction module* mentioned earlier which is required to enable a simulation to run.

observer.py

```
1  #!/usr/bin/env python2
2  import sys
3  import time
4  import SOAP
5
6  SOAP.Config.BuildWithNoType = 0
7  SOAP.Config.BuildWithNoNamespacePrefix = 0
8
9  server = SOAP.SOAPProxy("http://localhost:8000","http://dalt")
10
```

```
11 map_size = server.getMapSize() #get map dimensions
12 print "Map size is: ", map_size[0], ", ", map_size[1]
13
14 entities = server.getMapState() #get the current environment state
15 print "mapState done; got ", len(entities), " entities"
16
17 while (1): # run the simulation forever
18     l = 1
19     while (l) : # attempt to step the simulation
20         l = server.stepSimulation()
21         time.sleep(0.6)
22     l =1
23     while (l) : # wait until the current step finished
24         l=not server.isIdle()
25         time.sleep(0.6)
26
27     entities = server.getDelta() # retrieve the changes in the environment
28     print "getDelta done; got ", len(entities), " entities"
29
30     # process entities, display, etc...
```

observer.py

The call to `getMapSize` at line 11 causes the server to load the environment map, if it did not do so already. The main loop (starting line 17) follows the following steps:

- Try to run the next simulation step. If the server refuses to begin the next step, wait for 600ms and try again until a new cycle is launched;
- If the server is not idle yet (i.e. it is still processing a cycle) then wait for 600ms and check the state of the server again until the server finished processing its cycle;
- When the server finished processing the cycle we can retrieve all transitions that occurred in the last processing cycle. The `getDeltaMethod` provided by the server returns a list of entities representing the final state of the transitions that have occurred. As the observer has its own model of the environment, the whole transition can be reconstituted by computing the “difference” between the current state of each entity and the updated state obtained from `getDelta`;

- The whole cycle is repeated.

This observer implementation is mainly used for testing and debugging the rest of the toolkit. It has quicker startup times and is more easily modifiable due to its small size than the more complex observer described in the next section.

2.4.1.2.2 Complex observer An observer based on the design given in figure 2.10 on page 65 was implemented in the Java programming language. Java was chosen because of the ease and rapidity of developing applications with a graphical user interface.

The implementation of the Java observer follows the design closely, using the algorithms described above. The Java observer offers the same basic functionality as the Python observer, adding a two dimensional graphic display of the simulation environment.

The Java observer allows for users to run just one step of a simulation by executing a single call to the `stepSimulation` method on the server. Continuous running is implemented by employing a separate thread of execution posting `stepSimulation` requests continuously. Continuous running can be interrupted by signalling the thread to destroy itself.

In order to be able to display a meaningful representation of the environment, the observer maps each agent type to a colour, which is used when displaying that agent (the `EntityType` and `MapEntities` classes in figure 2.10 on page 65). The observer should work for most 2-d simulations, after the agent types used by the simulation are mapped to specific colours.

This is accomplished by adding statements of the form

adding Agent type ↦ Colour mappings

```
1 //display Food agents using green
2 types.add((Object) new EntityType("Food",Color.green));
3 //display Rabbit agents using grey
4 types.add((Object) new EntityType("Rabbit",Color.grey));
```

to the constructor for the class `MapEntities`.

In future, these mapping will be loaded from XML files, so modifying the source code for the observer will no longer be necessary.

2.4.2 The map editor

The role of the map editor is to manipulate XML files containing information about the environment characteristics (number of dimensions, and the span in each dimension) together with an initial layout for the agents involved in the simulation.

2.4.2.1 Map file format

The format of the XML file is fully described by the following DTD (Document Type Definition):

DALT map DTD

```
<?xml version="1.0"?>

<!-- the map has 0 or more entities and at least one dimension -->
<!ELEMENT map (entity*, axis+)>
        <!ATTLIST map dimensions CDATA #REQUIRED>

<!ELEMENT axis EMPTY>
        <!ATTLIST axis id CDATA #REQUIRED>
        <!ATTLIST axis value CDATA #REQUIRED>

<!-- each element has a position in the world specified as at least one dim -->
<!ELEMENT entity (pos+)>
        <!ATTLIST entity type CDATA #REQUIRED>
        <!ATTLIST entity colour CDATA #IMPLIED>

<!-- dimension has an axis is and the location -->
<!ELEMENT pos EMPTY>
        <!ATTLIST dim id CDATA #REQUIRED>
        <!ATTLIST dim value CDATA #REQUIRED>
```

DALT map DTD

2.4.2.2 Design

The map editor offers a facility of specifying the number of axis, together with the span on each axis. An empty environment is created, in which the experimenter can add and remove entities of different types. Maps can be saved, loaded and modified.

The UML design for a map editor handling two dimensional maps is given in figure 2.11.

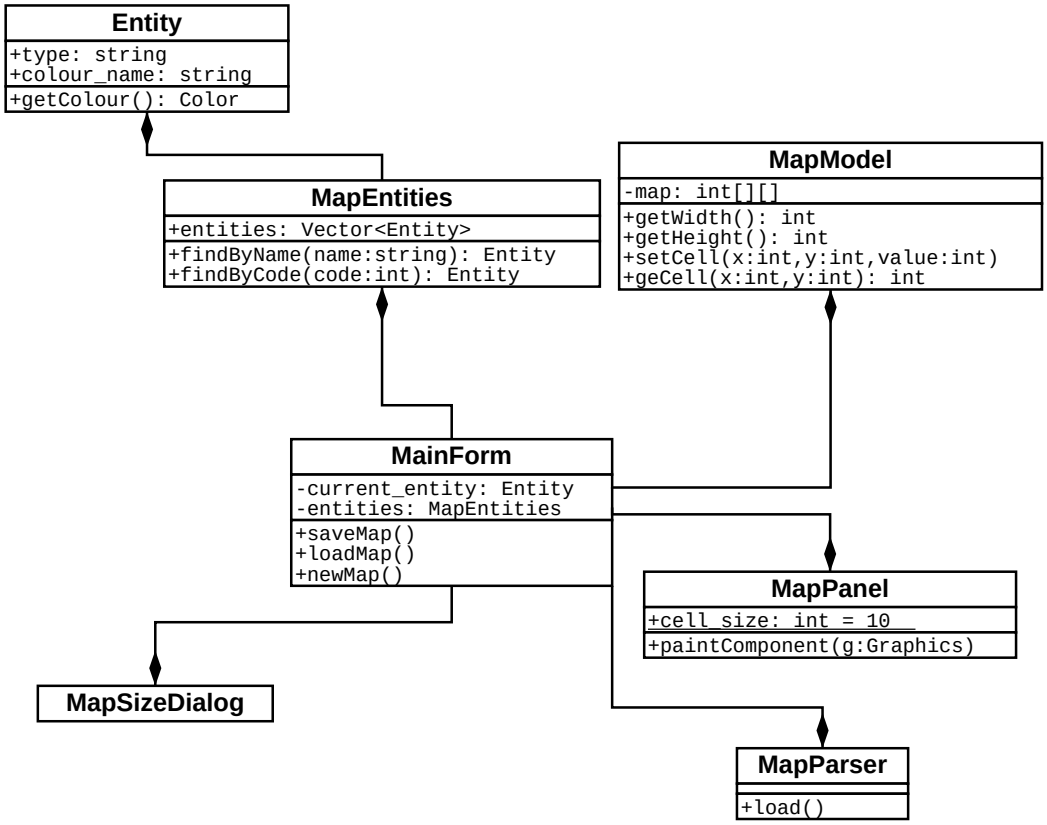

Figure 2.11: UML model for 2D map editor

2.4.2.3 Implementation

The map editor is implemented in the Java programming language, for the same reasons as the observer. The program uses the Java version of the *Xerces* library to parse and generate XML files. The design given in figure 2.11 is followed closely by the implementation. From a technical point of view, the implementation of this tool was straight forward — no special algorithms were employed and no major issues were encountered.

The only point worth mentioning is that, in a similar fashion to the observer, the map editor does not support external mappings from the agent types to different representation colours. In order to add/modify these mappings, the constructor for the class `MapEntites` has to be modified, by adding/removing entries of the form:

adding Agent type $\mapsto$ Colour mappings

```
1 //display Food agents using green
2 entities.add((Object) new EntityType("Food",Color.green));
```

```
3 //display Rabbit agents using grey
4 entities.add((Object) new EntityType("Rabbit",Color.grey));
```

The new agent types entries also have to be added to the menu. For example, in order to be able to use the new "food" agent type the following lines would have to be added to the `initComponents` method of the `MainForm` class:

adding menu entries

```
1 food_item = new javax.swing.JMenuItem();
2 food_item.setForeground(Color.green);
3 food_item.setText("Food");
4 food_item.addActionListener(new ActionListener() {
5     public void actionPerformed(ActionEvent evt) {
6         current_entity = entities.findByType ("Food");
7     }
8 });
```

In the future, these mappings will be read from an XML file and menus will be generated automatically. It was considered acceptable not to implement this feature as for each simulation type the modification will have to be performed only once when the simulation is set up and the time required to operate the modification is negligible.

2.5 Building simulations using DALT

This section shall describe the process which needs to be followed in order to build artificial life simulations using DALT.

The first step is to identify all the agent types which are going to be involved in the simulation. The observer and the map editor should be modified in order to support these agent types, following the procedure described in sections 2.4.2 and 2.4.1. The map editor and the observer may require extra modifications if the simulation is not using a 2-dimensional environment.

Next step involves identifying all the senses that the agents can use. Implementing the senses requires the following steps:

1. adding the sense definition to the client library, by extending the `Sense` class

2. implementing the sense functionality in the server library. This is accomplished by extending the `DALTServer` class and overloading the `soap_scanSense` method to handle the new sense

3. implementing the new sense may require changing the level of detail which is *observable* (see section 2.3.1.2.2 on page 41). In order to do this the following classes have to be altered:

   - the `Entity` class in the server
   - the `Entity` class in the client
   - the `SOAPEntity` class used by the server and the client. This class contains the `Entity` details which can be transfered between the server and the client, together with rules for marshaling/de-marshaling this information

Now the action requests should be identified and implemented. Once all the possible actions have been identified, the classes representing action requests have to be created by extending the `Action` class in the client library. The action request also includes constraints set on the originator and the target of the action (see section 2.3.1.2.3 on page 42). The actions themselves will be implemented later. Simulations may also need to modify the way in which action requests are arbitrated. This is accomplished by extending the `ActionArbitrage` class in the server and overloading the `processSuppliedActions` and `processRejectedActions` methods.

Separate class definitions are needed for each agent type. The classes should extend the `Agent` class from the client library. Each agent class should overload the `senseAndAct` method with their own method, doing all the core agent processing. Once these classes are built, the action execution can be implemented. For each action request that can target one agent, the `executeAction` method of that `Agent` has to be updated to include all the code which deals with performing that action.

The client side of the simulation is finalised once a class extending the `DALTClient` class is created. The class should contain the following:

- a `createAgent` method which handles the creation of any agent with a know type;

- a `soap_actionRequest` method which handles the conversion of SOAP messages carrying action requests into objects which are passed to the correct agents;

In order to finalise the server side of the simulation, a `serverCycle` method can be added to the class extending the `DALTServer` class. This method should contain all the custom processing that is to be carried out by the server at each cycle.

The observer will have to be modified further if the amount of observable data needs increased. The `Entity` class in the observer will have to be extended to include the new observable attributes. The experimenters may also wish to extend the observer in order to display other simulation data.

The process of building a simulation using DALT is illustrated with a practical example throughout the next chapter.

Chapter 3

Case study: game of life

This chapter will describe the design and implementation of a simple simulation, using DALT. Even though the simulation itself relativelly uncomplicated, it is designed such that it makes use of most of the features provided by DALT.

At the end of these chapter a range of experimental results are provided, showing the improvements obtained by using DALT.

3.1 The problem

The *Game of life* was invented by John H. Conway and was first published in the April 1970 issue of *Scientific American*. The game is played on a field of cells, each cell having 8 neighbours. Each cell can either be occupied by an "organism" or not. At each step the entire field is updated according to the following rules:

- an organism dies if it has less than 2 neighbours due to loneliness;
- an organism dies if it has more than 3 neighbours due to overcrowding;
- if an organism has 2 or 3 neighbours, it survives;
- if a cell of the field has 3 neighbouring organisms and is unoccupied, a new organism is born in that cell.

The problem is implementing a distributed simulation of the *Game of life*.

As the organisms are not “computationally intensive” we shall allow for an arbitrary amount of computation to be performed by each organisms at each simulation step in order to pose greater demands on the hardware used to run the simulation.

3.2 Analysis and design

The simulation environment is a 2-dimensional world, populated by organisms. As all organisms follow the same pattern, they can all be simulated by the same agent type. We shall call this agent type a *cell*.

The straight forward approach would be to analyse the state of the environment and decide which cells should die, which ones should stay alive, and which locations should contain new cells. In order to demonstrate the features of DALT we shall take a different approach.

In our simulation each *cell* has the capacity to “sense” its neighbours. Depending on the number of *cells* adjecent to it, the cell will kill itself or will stay alive.

During each simulation cycle, the entire environment will be analysed and new cells will be created in the locations which have 3 neighbours.

This approach demonstrates how to use agents, senses, actions and the server processing feature. These features are the main building blocks for DALT simulations.

3.3 Implementation

The implementation will follow closely the process recommended in section 2.5.

We shall start by modifiying the observer and the map editor to handle *cell* agents. The process described in the sections 2.4.1 and 2.4.2 is followed closely, mapping *cell* agents to the red colour. No further modifications are required.

Now we have a working map editor as it can be seen in figure 3.1 on the next page. The layout in figure 3.1 on the facing page, would result in following XML file:

map.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <map dimensions="2">
3     <axis id="0" value="100"/>
```

```
    <axis id="1" value="100"/>
    <entity type="cell" colour="red">
        <pos id="0" value="7"/>
        <pos id="1" value="9"/>
    </entity>
    <entity type="cell" colour="red">
        <pos id="0" value="7"/>
        <pos id="1" value="19"/>
    </entity>
.......
    <entity type="cell" colour="red">
        <pos id="0" value="26"/>
        <pos id="1" value="19"/>
    </entity>
</map>
```

map.xml

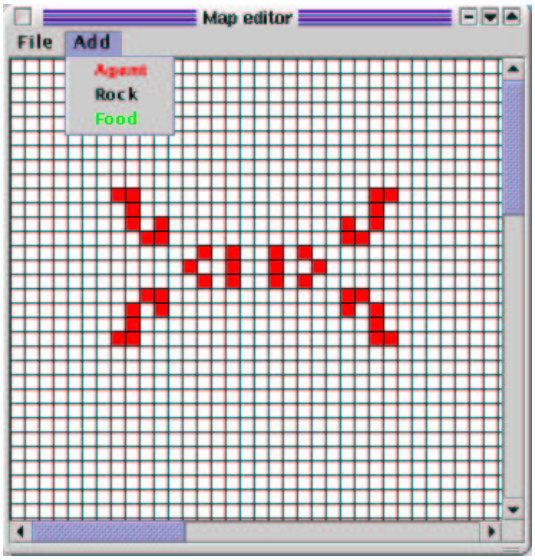

Figure 3.1: The map editor

A new sense, called `SenseNeighbours` is created by extending the `Sense` class in the client library:

SenseNeighbours

```
class SenseNeighbours : public Sense
{
public:
```

```
4       SenseNeighbours(SOAPProxy* endpoint): Sense ("neighbours",1, endpoint)
5         {};
6   };
```

SenseNeighbours

This definition simply creates a sense with the name “neighbours”, which takes 1 cycle to execute.

In order to implement the sense functionality, the `DALTServer` class is extended to create the `CAServer` class. In the `CAServer` class we overload the `soap_scanSense` method. The new method handles the “neighbours” sense we have created by returning all the *cells* adjacent to the *cell* originating the sense:

implementing the ‘‘neighbours’’ sense

```
1   if(sense == "neighbours") {
2       //first find the entity
3       Entity* e = wm->getEntityById(id);
4       //our rezult
5       SOAPArray<SOAPEntity> rez;
6
7       //now get the coord
8       int x = e->coordinates[0];
9       int y = e->coordinates[1];
10      for (int i=x-1;i<x+2;i++)
11          for (int j=y-1;j<y+2;j++)
12              if (i>=0 && j>= 0 && ! (x==i && y==j) && i<wm->axis[0] && j<wm->axis[1]) {
13                  vector<int> a;
14                  a.push_back(i); a.push_back(j);
15                  int n_id = wm->getCell(a);
16                  if( n_id !=0) {
17                       rez.Add(SOAPEntity(*wm->getEntityById(n_id)));
18                   }
19              }
20      response.AddParameter("sensed_entities") << rez;
21  }
```

implementing the ‘‘neighbours’’ sense

The cells will be able to perform only one action, KILL. Each cell will be able to post a kill request addressed to itself. All the requests get approved. The KILL action request is defined by extending the `Action` class in the client library, as follows:

KillCellAction

```
1  class KillCellAction : public Action
2  {
3  public:
4      KillCellAction(SOAPProxy* endpoint) : Action("kill_cell", endpoint, 1)
5      {
6          AllowedSets::Pair p;
7          p.sources.push_back("cell");
8          p.targets.push_back("cell");
9          allowed_sets.addPair(p);
10     }
11 };
```

KillCellAction

Lines 6–9 add the constraint specifying that cells can only kill other cells. As by default the `ActionArbitrage` class lets all action requests be executed and we shall make sure that agents do not attempt to kill other agents but themselves. There is no need to modify the `ActionArbitrage` class

We need to define a class specifying a *cell* agent:

definition for *cell* agent

```
1  class Cell : public Agent
2  {
3  public:
4          Cell(int id, CAClient* ca) : Agent("cell",id,ca) {}
5          void senseAndAct();
6          int  executeAction(Action& action);
7  };
```

The `Cell` agent is defined by extending the `Agent` class in the client library and specifying the name of the new agent type: in our case, “cell”. The `senseAndAct` is the main function of the agent. This is were `Cells` examine how many neighbours they have and decide whether to kill themselves or stay alive:

implementing senseAndAct()

```
1  vector<Entity> rezult;
2  agent_sync.acquire();
3  rezult = senses[0]->dispatch(); //use the first and only sense
```

```
4  agent_sync.release();
5
6  int a = 1;
7  for (int i=0;i<1500;i++)
8      for (int j=0;j<1000;j++) {
9        a++; a*=i; a/=i+1; a*=j; a/=j+1;
10       if (a>10000000) a= 5000000;
11     }
12
13 if (rezult.size() != 3 && rezult.size() !=2) {
14     //create a kill cell action
15     KillCellAction act = KillCellAction(dalt_client->endpoint);
16     act.setOriginator(dynamic_cast<Entity*>(this));
17     act.addTarget(*(dynamic_cast<Entity*>(this)));
18     try {
19         agent_sync.acquire();
20         act.dispatch();
21         agent_sync.release();
22     } catch(Synchronization_Exception& e) {
23         cout << "Synchronization exception: "<<e.what() <<endl;
24     }
25 }
```

At line 3 the `Cell` uses its `SenseNeighbours` sense. Note that all statements which involve dispatching SOAP messages need to be protected by mutual exclusion zones (lines 2,4). In lines 6–11 the cells executes some code which should slow down the execution of each cell, in order to simulate the behaviour that a complex agent would have. This is necessary in order to shift the speed bottle-neck from communication to processing. If this code is commented out the simulation would only be limited by the speed at which the SOAP messages can be processed. With the code in place, the simulation is limited by processor speed rather than communication bandwidth.

Finally, if the sense did not return 2 or 3 entities (line 13) the agent will dispatch a `KillCellAction` request (lines 15,20) from itself (line 16) to itself (line 17).

In the implementation of the `executeAction` function the `Cell` agent has to be able to handle

incoming action request, which in our case can only be of the type `KillCellAction`

implementing executeAction

```
1  if(action.name == "kill_cell") {
2    state_change_method->AddParameter("initial_state") << SOAPEntity(*this);
3    SOAPEntity final;
4    final.entity_type=""; //set type to empty string - this is equivalent to a null entity
5    state_change_method->AddParameter("final_state") << final;
6
7    try {
8      agent_sync.acquire();
9      dalt_client->endpoint->Execute(*state_change_method);
10     agent_sync.release();
11     dalt_client->killAgent(id); //now queue for destruction
12   } catch(Synchronization_Exception& e) {
13     cout << "Synchronization exception: "<< e.what() <<endl;
14   }
15 }
```

If the agent is about to die it has to notify the server, as this would result in a change in the observable state of the agent (i.e. the agent disappears form the environment completely). This is accomplished in lines 1–10. The class `DALTClient` provides the `killAgent` method which destroys the agent. This method is used by the agent to destroy itself in line 11.

Next we need to extend the existing `DALTClient` class. A new `CAClient` class is created, capable of handling the creation of `Cell` agents forwarding `KillCellAction` requests:

the CAClient class

```
1  class CAClient : public SOAPDispatchHandler<CAClient>, public DALTClient
2  {
3  public:
4  CAClient(int port_no, string server);
5
6  ///returns pointer to this instance of the class
7  virtual CAClient* GetTarget(const SOAPEnvelope& request) { return this; }
8
9  int createAgent(string type, int id)
10 {
```

```
11    if(type == "cell") {
12      Cell* a = new Cell(id, this); //init the agent
13      SenseNeighbours* sense_n = new SenseNeighbours(endpoint);
14      a->addSense(sense_n); //add senses
15      agents.push_back(a); //add agent to list
16      a->start(); //start it
17      return 0;
18    } else {
19      cout <<"Agent  of type *"<<type<<"* unknown\n";
20    }
21    return 1;
22  }
23
24  void soap_actionRequest(const SOAPMethod& request, SOAPMethod& response)
25  {
26    SOAPString action;
27    int id;
28    SOAPArray<int> targets;
29    request.GetParameter("entity_id") >> id;
30    request.GetParameter("action") >> action;
31    request.GetParameter("targets") >> targets;
32
33    if(action=="kill_cell") {
34      for(unsigned int j=0;j<targets.size();j++)       //for each target
35        for(unsigned int i=0;i<agents.size();i++) //find the agent
36          if(agents[i]->id == targets[j]) {
37            KillCellAction* kc = new KillCellAction(endpoint);
38            agents[i]->queueAction(kc); //and dispatch the action
39            break;
40          }
41      response.AddParameter("performed") <<0;
42      return;
43    }
44    response.AddParameter("performed") <<1;
45  }
46  };
```

The `createAgent` method, creates a new `Cell` object, adds the `SenseNeighbours` sense to its pool of available senses and launches it by `start()`ing the thread which support the `Cell`.

The `soap_actionRequest` method identifies each cell targeted by the action request and creates a new `KillCellAction` object which is queued for execution by each target.

In order to complete the implementation for the *Game of life* the `serverCycle` method needs to be implemented within the `CAServer`. This method will examine the current state of the environment and create new `Cells` in the locations which have 3 neighbours:

implementing the server cycle

```
1  vector < vector<int> > to_create;
2  for(int x=0;x<wm->axis[0];x++)
3    for(int y=0;y<wm->axis[1];y++) {
4      int neigh = 0;
5      vector<int> v;
6      v.push_back(x); v.push_back(y);
7      if (wm->getCell(v) != 0) continue; //skip if this cell is alive
8      for (int i=x-1;i<x+2;i++)
9        for (int j=y-1;j<y+2;j++)
10         if (i>=0 && j>= 0 && (x!=i || y!=j) && i<wm->axis[0] && j<wm->axis[1]) {
11           vector<int> a;
12           a.push_back(i); a.push_back(j);
13           if( wm->getCell(a) !=0)
14             neigh++;
15         }
16     if(neigh == 3) { //create new cell here
17       vector<int> t;
18       t.push_back(x); t.push_back(y);
19       to_create.push_back(t);
20     }
21   }
22 for(unsigned int i =0;i<to_create.size();i++) {
23   Entity* e = entity_factory.createEntity("cell",to_create[i],wm->all_entities);
24   e->is_ready = true;
25   wm->stateChange(NULL, e); //signal state change (update the environment)
26   allocateEntity(*e);
27 }
```

The functions parses the entire environment by using the representation provided by the `WorldMap` class (accessed through the `wm` object) starting with the lines 2 and 3. If a location is found that could host a new `Cell`, the cell cannot be created immediately as it would influence the outcome of the tests for the neighbouring locations. The cell is instead queued for creation in the `to_create vector`.

For each location not containing a cell already (line 7) all its 8 neighbouring locations are checked and the neighbours are counted (lines 8–15). If the location has 3 adjacent neighbours, the location is queued for receiving a new cell (lines 16–20).

When the entire environment has been processed, new `Cells` are created for each location queued in the `to_create` vector. The `EntityFactory` is used to allocate each new `Entity`. The `WorldMap` object is notified by the change of the observable state within the environment (apparition of a new cell) in line 26. Finally, the `Entity` is allocated to the best suited client in line 26.

3.4 Results

Once the implementation is finished the simulation can be ran. The observer will continuously display the current state of the environment and will allow the experimenter to control the simulation, as shown in figure 3.2.

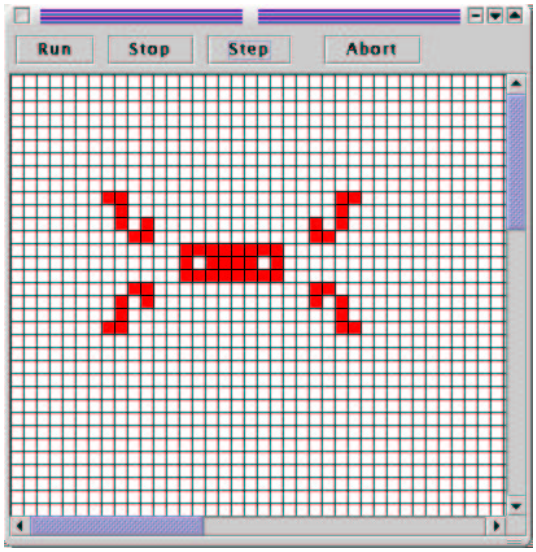

Figure 3.2: Observing the simulation

The system obtained can be used to run a series of tests in order to investigate the efficiency of the toolkit. All the tests detailed below were run using two computers of different powers linked by a 10Mb Ethernet network. The maximum throughput of the network was around 600Kb/second. The machines have the following specification:

machine 1 (ra) AMD Duron 700Mhz processor, 512 Mb RAM

machine 2 (nusku) Intel Pentium III 933Mhz processor, 256 Mb RAM

The test data for the simulation was an environment containing cells laid in an oscillator shape. The cellular automata goes through a set of different configurations, creating and destroying some cells and periodically returning to the initial configuration and repeating the cycle.

3.4.1 Light processing

The first experiment ran is going to investigated the way the performance modifies when a distributed simulation with agents doing very little computation is ran. In order to execute this experiment, the code within the `Cell` class which stresses the processor was commented out. The timing information over 100 cycles can be seen in figure 3.3. The timing information plotted is the

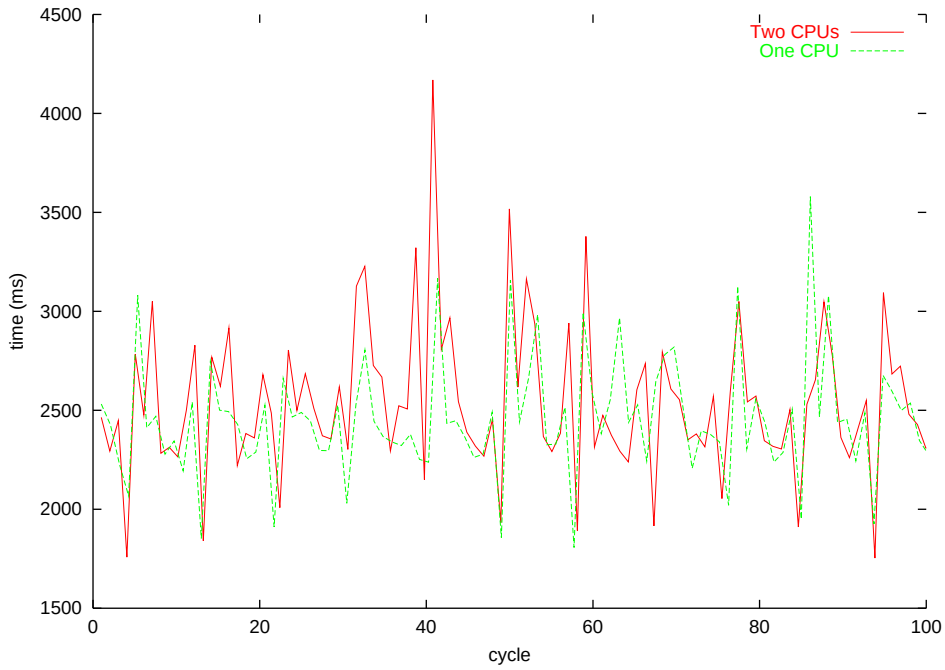

Figure 3.3: Performance for non-computationally intensive simulations

time taken to compute a cycle. The performance deteriorates when the simulation is run on more than one machine. Further investigation revealed that the entire network bandwidth was used, while only part of the available processing power was employed. The SOAP server accepted requests at maximum rate.

The performance impact on simulations doing light processing can be more clearly seen in figure 3.4, which plots the average cycle time at each step of the simulation.

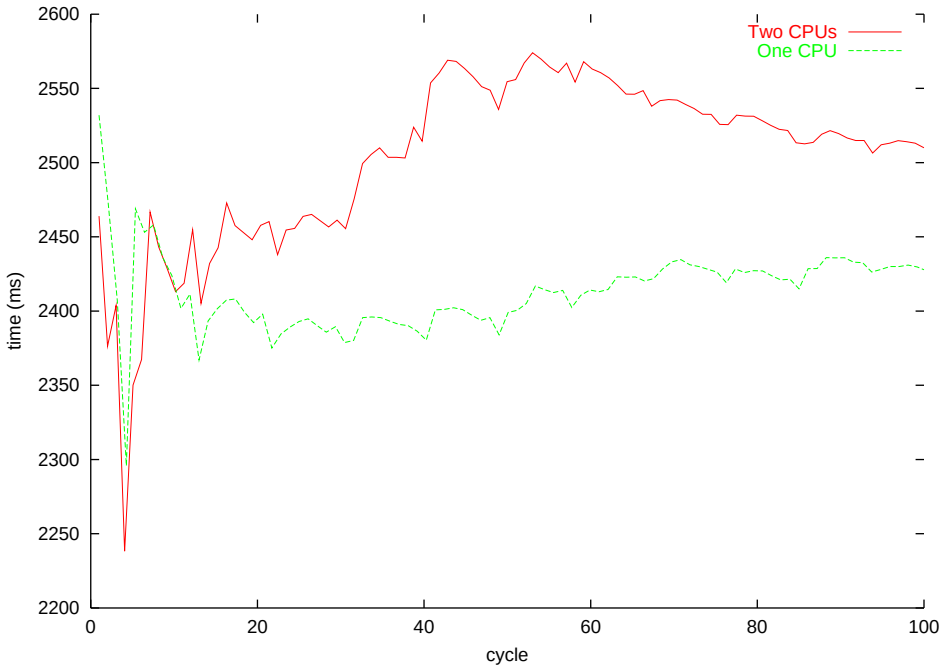

Figure 3.4: Average cycle time for non-computationally intensive simulations

It is worth mentioning that when the simulation is running on one machine only, all the communication is carried out via the loopback interface, which is only limited in speed and bandwidth by the available processing and memory within the host system.

3.4.2 Intensive processing

The simulation was launched again on the same machines, after enabling the code which is stressing the processor. The performance of the toolkit in this case can be seen in figure 3.5 on the facing page.

In this case the average cycle time for the distributed simulation is up to 45% lower. This can be seen more clearly in figure 3.6 on page 86 which plots the average cycle time at each step.

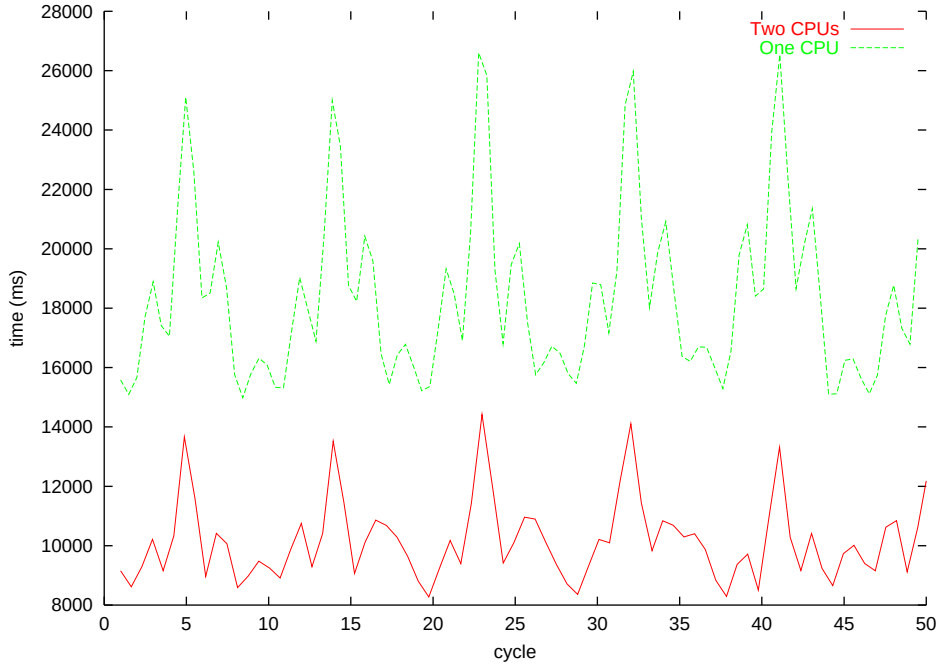

Figure 3.5: Performance for computationally intensive simulations

The simulation is limited in speed by the processor power and the speed at which the server can accept incoming connections.

3.4.3 Allocation algorithms

The performance of the three different allocation algorithms supplied by DALT is illustrated in figure 3.7 on page 87. Each graph shows the difference between the time of the slowest client and the time of the quickest client. The better performing algorithms minimise this difference. It is apparent that the greedy algorithm is the best performing one, followed by weighted random allocation. The worst performing algorithm in this case is the plain random allocation.

However, different simulations might find that different algorithms work better. For example, if all the workstations have the same processing power (which is not uncommon, as institutions usually purchase several computers with identical specifications at once) the plain random algorithm should perform slightly better than the weighed random algorithm, as the server requires less computational power to produce similar results.

It is entirely possible that for some simulations there is no similarity between the time taken to execute agents of the same type. In these cases it is very likely that the random allocation

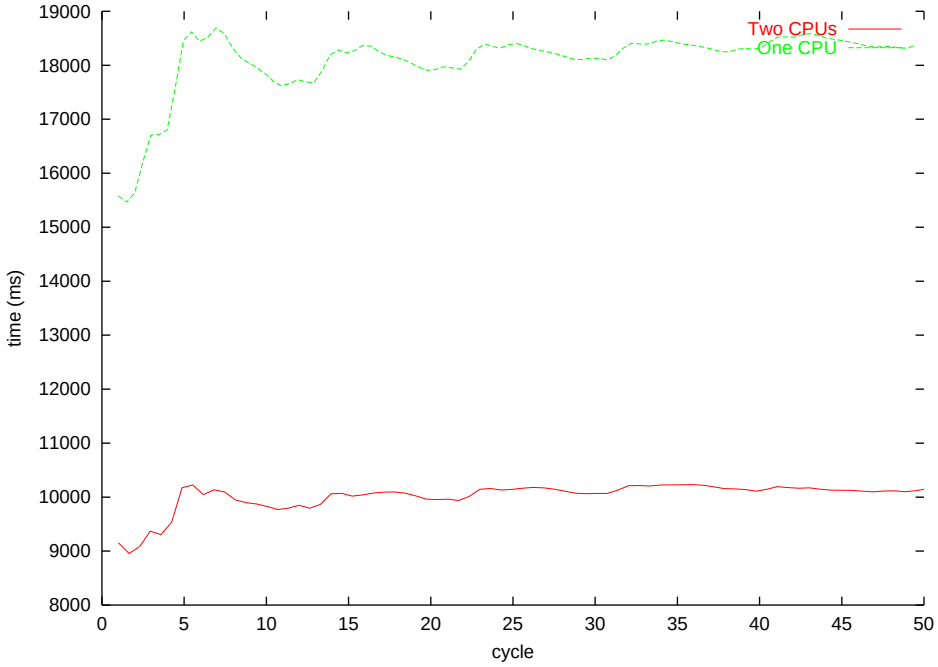

Figure 3.6: Average cycle times for computationally intensive simulations

methods would perform better than the greedy allocation.

3.4.4 Benchmark weights

Finally the impact produced by modifying the weights of different machine performance statistics is analysed (see section 2.1.1.8 on page 34). The graph labelled “equal distribution” plots the time taken by the simulation to complete each cycle, when the memory operations, integer operations and floating point operations are given the same weight in benhmarking the machines.

As in the “game of life” simulation we use mostly integer operations, we modified the weights such that the integer performance accounts for 80% of the machine performance, memory operations account for 19% and floating point operation for 1%. The simulation was run again for the same data set. The graph for this distribution is labelled “weighed distribution” (see figure 3.8 on page 88

The difference between the respective simulation speeds is not great. Surprisingly, the simulation ran with “equal distribution” benchmarking performs slightly better than the other one. Further investigation revealed the machines running the simulation were quite close in processing power: one of them was indexed (evenly) at 4.08 and the other one at 5.12. The weighed performance

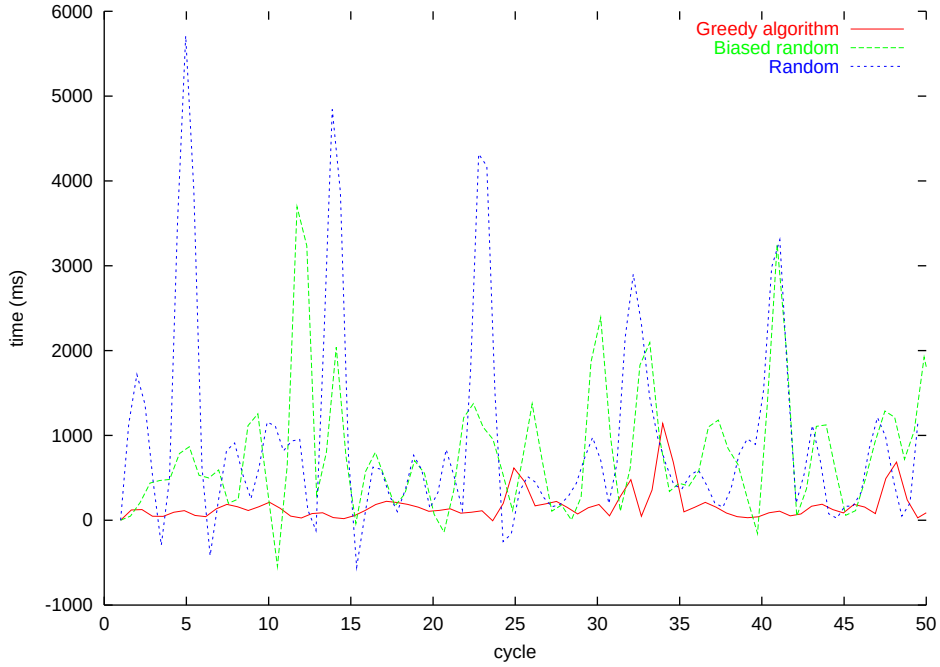

Figure 3.7: Performance of different allocation algorithms

indexing resulted in a figure of 2.82 for the first machine and 3.51 for the second. The difference in proportional performance is quite possibly too small to make a difference in allocation: 1.44 vs 1.45. However, as the actual figures for evenly distributed benchmark are greater than the figures for the weighted distribution, they are likely to enable the allocation algorithm to make more accurate predictions. This is likely to be the reason for which the obtained results do not coincide with the expected ones in this case.

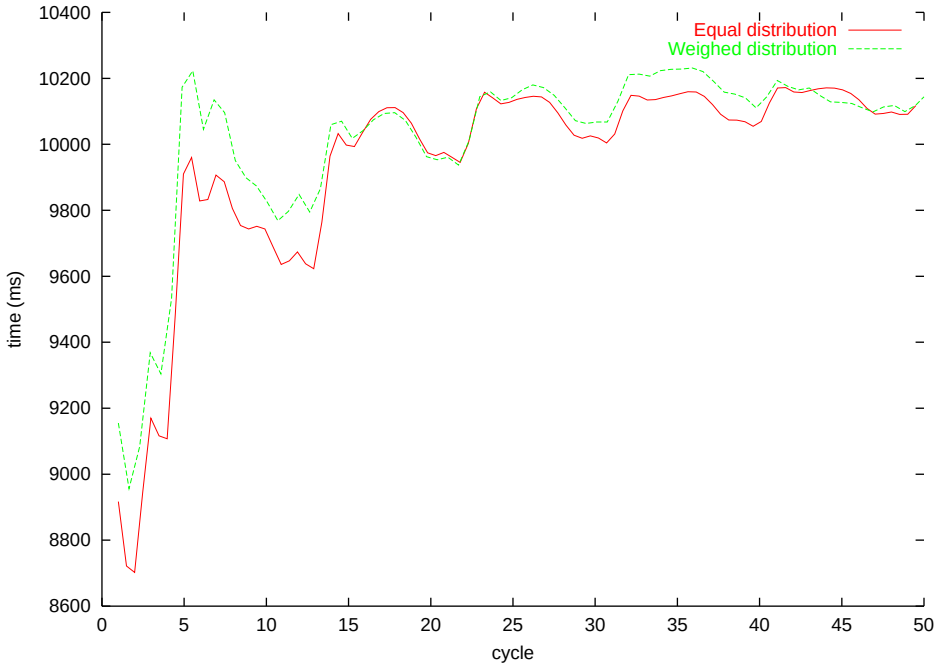

Figure 3.8: Impact of different benchmark statistics distributions

Chapter 4

Discussion

4.1 Project goals

All the goals of the project have been attained. The toolkit was created as initially stated and it was used to construct a test simulation. The test simulation was obtain with relatively little implementation effort and provided a good way of testing the toolkit.

The testing revealed that running the simulation in a distributed fashion can greatly improve the overall speed of the simulation, which was exactly the goal the project was aiming for,

There has not been any testing done with more than two machines, mainly due to the time required to run such a simulation. The probable performance of the toolkit on several machines can be inferred from the tests that were accomplished and the architecture.

The modularity and language independence goals were achieved by using SOAP, a flexible, machine-independent communication protocol which has implementations available for most major programming language.

The toolkit itself is written in portable C++. The platform dependent features have been isolated and are handled by separate configuration tools. Some tools are written in Java and Python, which are both platform independent languages.

4.2 Conclusions

It is possible to speed up existing and future artificial life simulation by using DALT to distribute the simulation across a network.

The case study given in chapter 3 on page 73 covers the main features which are likely to be used by a simulation based on DALT. Using the toolkit gives a noticeable improvement, which can go up to 45% by adding only an extra machine, as demonstrated in section 3.4 on page 82.

It is important that the physical network supplying the communication infrastructure has a large bandwidth and low latency in order to speed up the communication in between different computation nodes and the server. The machine running the server part of the simulation needs to have significant processing power in order to be able to deal with incoming requests as quickly as possible.

Although the SOAP communication protocol works perfectly, the current library used to carry out SOAP communication is flawed, slowing down the speed at which the incoming messages can be accepted and processed. The performance could be improved by using a different library.

DALT provides the libraries and tools necessary for building distributed simulations and provides a flexible framework which can be easily adapted to suit a variety of different requirements.

4.3 Practical applications

DALT offeres a starting point and a methodology for implementing distributed artificial life simulations. The inter-module communication is standardised and extensible. The agent design is left mostly up to particular implementations. Experiments using C, C++, Fortran, Prolog or any other programming language which can export “C style” functions can make use of this library to distribute their computation.

Simulations which contain computationally intensive agents will gain a significant increase in speed.

4.4 Difficulties

The project encountered several difficulties during its lifespan. First of all, it is difficult to find a central repository of information about artificial life simulations, which is up to date. There are several sites on the Internet attempting to catalogue this area, but most of them contain old information and links to dead projects. This situations rendered the task of verifying whether there has been any previous work done in the projects are quite difficult.

Once the design stage was complete the implementation proved to be a lot more time consuming than expected. Finding the right libraries to use in development was hard. There are many libraries, but only a few of them are stable enough for supporting a project like DALT. For example, three different threading libraries have been used with DALT and all of them either some vital functionality missing (such as the ability to join threads for example), or even worse, bugs burried deep within the library. This kind of bugs slowed down development immensily. Their shortcomings did not became apparent until the implementation was well on its way.

A key design issue which caused a lot of trouble is the parallelism of the simulation. This generates many possibilities for concurrency problems, such as deadlocking and sharing conflicts which were dealt with during the implementation process.

4.5 Lessons learnt

Designing the project thoroughly **and** investigating the tools which were to be used during the implementation before the implementation started was highly beneficial, greatly reducing development time. The main lesson learnt from the implementation process is never to fully trust the tools you are using. Assuming that my own implementation of some bit of code is faulty rather than trying to find the source of the problem within the supporting libraries costed many hours of waisted time, debugging perfectly good code.

4.6 Future work

DALT can be extended in several ways, to increase its performance and flexibility. One extension that can be implemented is allowing the clients to group all the passive agents onto one thread,

and process them at once. This should increase the speed of simulations that use many passive agents (such as rocks, tree, etc). It should also decrease the memory usage will possibly increase the overall speed of the simulation due a decrease in the amount of data that needs to be transferred on the network. This approach to handling passive agents is suggested in Servat et al. [1998].

The toolkit uses *NBench* for benchmarking. Switching to another benchmarking application might give more accurate performance numbers. A good improvement to support this would be using the simulation itself as for benchmarking.

The supporting tools (the map editor and the observer) can be improved to be able to handle the introduction of different types without the need for modifying the source code.

Another improvement which can be added to the toolkit is the ability to load and save entire simulations, to enable experimenters to stop a simulation and resume it at a later date. This would involve building into the toolkit automatic serialisation mechanisms, which would allow each and every agent to be stored on the server on the simulation shutdown.

At the moment the toolkit is fault intolerant. By implementing the saving/loading mechanism described above, a certain amount of fault recovery can be obtained. If a client is unreachable for example, the simulation can be reverted to the most recent saved state and the faulty client removed.

It would also be very interesting to see how the toolkit performs in a practical simulation, using a large number of computers on a high speed network.

Appendix A

DALT communication

A.1 Overview

The whole toolkit is based around the communication in between the server and the client. The communication protocol has to be language and architecture independent so clients can be easily implemented in various languages, or even a reimplementation of the server should be fairly easy.

SOAP makes an ideal protocol with regard to the issues mentioned above. It is open, it is based on XML and it has implementations for most of the popular programming languages. In order to make use of the features provided by SOAP [Seely, 2002, W3C, 2001] the messages and their content have to be clearly specified, acting as the main source of information for the main implementations and subsequent re-implementations.

All the modules of the toolkit will have to provide a SOAP server and a SOAP client. The SOAP server provides methods that can be invoked remotely by the clients. The clients can invoke remote methods on any of the known servers.

All the types described below are the ones specified in the `xsd` and `xsi` namespaces used by the *SOAP* specification. This leaves the task of serialising objects up to the individual module implementation, but ensures a consistent message format, which could not be easily achieved if the serialising task would be left up to an automatised process.

A.2 Services provided by the DALT server

A.2.1 Client registration

The server is the first program module of the toolkit which started. After start-up the server waits for clients to make their presence known by calling this method. The clients have to 'present' themselves by providing the contact details for their soap server. In exchange they will receive 0 or 1 depending on whether the server accepted the registration. There is no need to keep track of the clients with unique ids as the address of the client SOAP server is unique.

Method: *clientRegistration*

Parameters

- *client_endpoint* (`xsd:string`) — URL for the client SOAP server
- *client_ns* (`xsd:string`) — the name space used by the client
- *speed_factor* (`xsd:float`) — The benchmarked speed of the machine the client is running on

Returns

- *performed* (`xsd:int`) — 0 for approved, 1 for not not approved. Other values may be added later

A.2.2 Sense requests

The agents will periodically need to scan their available senses. While the agents are aware by the senses possessed the actual processing is done by server. The server makes available the following method:

Method: *scanSense*

Parameters

- *entity_id* (`xsd:int`) — this is the id of the entity (agent) requesting this sense.
- *sense* (`xsd:string`) — the name of the sense being used

Returns

- *sensed_entities* (`SOAP-ENC:Array, SOAP-ENC:arrayType="Entity"`) — an array with the sensed entities

Where *Entity* is a compound value containing the following:

- *entity_id* (`xsd:int`) — identifies the entity
- *entity_type* (`xsd:string`) — the type of this entity
- *coordinates* (`SOAP-ENC:Array, SOAP-ENC:arrayType="xsd:int"`) — an array with the position (with an origin in the current position of the agent originating the sense request) of this entity on all axes

Of course, this is only the base form of a sense. Custom senses may send more information to the server by using extra parameters, but the return values will always keep the same format. In order to include more "observable" information, the *Entity* compound can be extended to provide extra information.

A.2.3 Action request

The action request keeps a similar format to the sense request.

Method: *actionRequest*

Parameters

- *entity_id* (`xsd:int`) — this is the id of the entity (agent) originating the action
- *action* (`xsd:string`) — the name of the action being used
- *targets* (`SOAP-ENC:Array, SOAP-ENC:arrayType="xsd:int"`) — an array with the IDs of the entities that are specifically targeted by this action

Returns

- *performed* (`xsd:int`) — 0 for performed, 1 for not performed. Other values may be added later

Again, specific actions will build on this format by adding extra parameters to the action call. The *server* normally queues the action in an action queue, unless an immediate conflict is risen.

A.2.4 State change notification

Whenever an agent changes its observable state, the server is notified by this change so that the map (and indirectly the observer) can be updated.

The message contains two entities with the same format as the ones described in section A.2.2 on page 94. For position changes, the second entity will contain the relative position to the first entity. For creation/destruction one of the entities will be replaced with an entity with empty type. This method returns nothing.

Method: *stateChange*

Parameters

- *initial_state* (`Entity`) — the initial entity state
- *final_state* (`Entity`) — the final entity state

A.2.5 Cycle completed notification

Whenever an agent executed all the actions it could in a cycle and is ready for a new cycle a cycle complete message is sent to the server. The *cycleComplete* method provided by the server takes as parameter only the ID of the entity sending this notification. A value of 0 or 1 is returned depending on whether the server approves the request or not.

Method: *cycleComplete*

Parameters

- *entity_id* (`xsd:int`) — the ID of the entity notifying the completion of its cycle

Returns

- *performed* (`xsd:int`) — 0 for approved, 1 for not not approved. Other values may be added later

A.3 Agent to agent communication

All the inter-agent communication passes through the server (reasons for this are described in 2.3.2). A positive side-effect of this is that not every agent is required to perform as a SOAP

server, which would add a computational burden for simulations with many agents. All the action requests are sent to the server which are in turned re-routed to the right client. The client de-serialises the message and passes it internally to the right agent.

For some actions, the server might be able to perform the actions without forwarding, so there would be no need to pass the messages on.

A.4 Methods provided by the client

A.4.1 Client status

This method provides generic information about the client.

Method: *clientStatus*

Parameters

- none

Returns

- *status_info* (`ClientInformation`) — a compound containing the requested information

The *ClientInformation* compound contains the following:

- *last_cycle_time* (`xsd:int`) — time taken to complete last cycle
- *average_cycle_time* (`xsd:int`) — average time taken to complete a cycle

More fields may be added to this compound.

A.4.2 Agent status

This method provides generic information about an agent.

Method: *agentStatus*

Parameters

- *entity_id* (`xsd:int`) — the id of the agent to which the message is to be routed

Returns

- *status_info* (`Array:AgentInformation`) — a compound containing the requested information

The *AgentInformation* compound contains the following:

- *last_cycle_time* (`xsd:int`) — **normalised** time taken to complete last cycle
- *average_cycle_time* (`xsd:int`) — average **normalised** time taken to complete a cycle

More fields may be added to this compound.

A.4.3 Agent creation

This method is used by the server to create a new agent. The only information supplied is the type of the agent. For more complex simulation more information might be provided, such as an initial internal set of states for the agent or cloning information.

Method: *createAgent*

Parameters

- *type_id* (`xsd:int`) — the type of the agent to be created
- *agent_id* (`xsd:int`) — the ID of the agent to be created (allocated by the server)

Returns

- *performed* (`xsd:int`) — 0 for approved, 1 for not not approved. Other values may be added later

A.4.4 New cycle notification

When this method is called a client notifies all its agents of the beginning of a new cycle.

Method: *newCycleNotify*

Parameters

- none

Returns

- none

A.4.4.1 Action request

The interface for the *ActionRequest* method is identical to the one used by the server (section A.2.3 on page 95). The client de-serialises the information provided and passes the request to the appropriate agent.

A.5 Server control

The server provides the following methods for invocation by the observer:

A.5.1 Stepping the simulation

Method: *stepSimulation*

Parameters

- none

Returns

- *rezult* (`xsd:int`) — 0 if a new cycle was initiated, a positive value greater then 0 otherwise

When the observer calls this method, the server attempts to execute one step of the simulation. This will happen only if all the entities have finished the previous cycle and are ready to start. If all entities are ready, a new cycle is started and the method will return the value 0. Otherwise a new cycle is *not* started and a positive value is returned.

A.5.2 Retrieving the entities modificated created in the last cycle

Method: *getDelta*

Parameters

- none

Returns

- *changed_entities* (`SOAP-ENC:Array, SOAP-ENC:arrayType="Entity"`) — an array with the entites which this simulation cycle changed

All the entites which have suffered visible changes during the last cycle are returned in the *changed_entities* array. The observer should be able to distinguish in between new entities (entities whose uniques IDs have not been encountered in previous steps), entities which simply changed their state (known IDs) and entites which have been destroyed (indicated by an empty entity type).

A.5.3 Testing server state

Method: *isIdle*

Parameters

- none

Returns

- *rezult* (`xsd:boolean`) — true if the server is idle (and ready), false if the server is still running a cycle

This function has to be used to poll the server in order to determine when it is safe to retrieve the changes occured in the cycle. This also acts as a trigger inside the server for certain operations, such as delivering actions. It is very important that the *observer* actually uses this method.

A.5.4 Map status

This method is similar to the *stepSimulation* method. The main difference is that no action is performed by the server.

Method: *mapStatus*

Parameters

- none

Returns

- *entities* (`SOAP-ENC:Array, SOAP-ENC:arrayType="Entity"`) — an array with the entites in the world

Appendix B

Patching Easysoap

The most recent version of EasySoap[1] has some severe problems in the Abyss HTTP server. After extensive testing and debugging in turned out that the performance is very poor if the keep-alive mechanism is enabled. After a compilation with the default setting, both the server and inser the client will attempt to keep the TCP/IP connection alive, but they fail, run out of available here connections and need to wait for an existing connection to be closed. This causes frequent 15 seconds delays which make the program unrunnable. Debugging has shown that if the delay is decreased to over a second, the performance is still unacceptable. If the delay is decreased to under a second, some connections will not stay open long enough for the SOAP requests to complete and in consequence the requests will fail. The author has been quite unhelpfull with regard to this problem.

The only solution I managed to find is deactivating the keep-alive mechanism altogether, both in the *Easysoap* library and in the code for the *Abyss* web server. This leads to dramatic performance improvments. Below is the `diff` obtained from the source code. Use `patch -p1 < easysoap.patch` in order to apply.

easysoap.patch

```
diff -ur easysoap/src/SOAPWinInetTransport.cpp
        easysoap-orig/src/SOAPWinInetTransport.cpp
--- easysoap/src/SOAPWinInetTransport.cpp        Sat Jan 26 13:18:58 2002
+++ easysoap-orig/src/SOAPWinInetTransport.cpp        Thu Dec 20 17:38:19 2001
@@ -32,7 +32,7 @@
```

[1] which can be obtained via anonymous CVS access from http://www.sf.net/projects/easysoap

```
 USING_EASYSOAP_NAMESPACE

 SOAPWinInetTransport::SOAPWinInetTransport()
-: m_keepAlive(false)
+: m_keepAlive(true)
 , m_hInternet(NULL)
 , m_hConnect(NULL)
 , m_hRequest(NULL)
@@ -41,7 +41,7 @@
 }

 SOAPWinInetTransport::SOAPWinInetTransport(const SOAPUrl& endpoint)
-: m_keepAlive(false)
+: m_keepAlive(true)
 , m_hInternet(NULL)
 , m_hConnect(NULL)
 , m_hRequest(NULL)
3etTransport::SOAPWinInetTransport(const SOAPUrl& endpoint,
        const SOAPUrl& proxy)
-: m_keepAlive(false)
+: m_keepAlive(true)
 , m_hInternet(NULL)
 , m_hConnect(NULL)
 , m_hRequest(NULL)
diff -ur easysoap/src/SOAPonHTTP.cpp easysoap-orig/src/SOAPonHTTP.cpp
--- easysoap/src/SOAPonHTTP.cpp        Sat Jan 26 13:18:00 2002
+++ easysoap-orig/src/SOAPonHTTP.cpp        Wed Dec 19 15:48:56 2001
@@ -238,8 +238,8 @@
         WriteLine(" HTTP/1.1");
         WriteHostHeader(m_endpoint);

-//        if (m_keepAlive)
-//                WriteHeader("Connection", "Keep-Alive");
+        if (m_keepAlive)
+                WriteHeader("Connection", "Keep-Alive");
 }
```

```
 void
@@ -369,7 +369,7 @@
         //
         m_canread = GetContentLength();
         m_doclose = false;
-/*        const char *keepalive = GetHeader("Connection");
+        const char *keepalive = GetHeader("Connection");
         if (respver > 10)
         {
                 if (keepalive && sp_strcasecmp(keepalive, "Keep-Alive") != 0)
@@ -381,7 +381,7 @@
                         m_doclose = true;
         }

-        if (!m_keepAlive)*/
+        if (!m_keepAlive)
                 m_doclose = true;
         //
         // Check if HTTP is Transfer Endoded: Chunked
diff -ur easysoap/src/abyss/src/abyss.h easysoap-orig/src/abyss/src/abyss.h
--- easysoap/src/abyss/src/abyss.h        Sat Jan 26 16:04:49 2002
+++ easysoap-orig/src/abyss/src/abyss.h        Tue Sep  4 20:46:53 2001
@@ -59,7 +59,7 @@
 ** Maximum numer of simultaneous connections
 ******************************************************************/

-#define MAX_CONN        1
+#define MAX_CONN        16

 /******************************************************************
 ** DON'T CHANGE THE FOLLOWING LINES
diff -ur easysoap/src/abyss/src/http.c easysoap-orig/src/abyss/src/http.c
--- easysoap/src/abyss/src/http.c        Sat Jan 26 13:53:50 2002
+++ easysoap-orig/src/abyss/src/http.c        Fri Jul 13 14:06:42 2001
@@ -288,8 +288,8 @@
```

```
                r->keepalive = FALSE;
                /* keepalive is default for HTTP/1.1 */
-               /*if (vmaj > 0 && (vmaj != 1 || vmin != 0))
-                 r->keepalive = TRUE;*/
+               if (vmaj > 0 && (vmaj != 1 || vmin != 0))
+                       r->keepalive = TRUE;
                r->versionmajor=vmaj;
                r->versionminor=vmin;
        }
@@ -348,9 +348,9 @@
                if (strcmp(n,"connection")==0)
                {
                        /* must handle the jigsaw TE,keepalive */
-                 /*                       if (strcasecmp(t,"keep-alive")==0)
+                       if (strcasecmp(t,"keep-alive")==0)
                                r->keepalive=TRUE;
-                               else*/
+                       else
                                r->keepalive=FALSE;
                }
                else if (strcmp(n,"host")==0)
diff -ur easysoap/src/abyss/src/server.c easysoap-orig/src/abyss/src/server.c
--- easysoap/src/abyss/src/server.c        Sat Jan 26 16:03:33 2002
+++ easysoap-orig/src/abyss/src/server.c        Thu Jan 10 13:05:13 2002
@@ -553,7 +553,7 @@
        srv->keepalivetimeout=15;
        srv->keepalivemaxconn=30;
        srv->timeout=15;
-       srv->advertise=FALSE;
+       srv->advertise=TRUE;
        srv->userdata=0;
        srv->stopped = 0;
 #ifdef _UNIX
diff -ur easysoap/src/abyss/src/threadpool.h
        easysoap-orig/src/abyss/src/threadpool.h
```

```
--- easysoap/src/abyss/src/threadpool.h        Sat Jan 26 16:00:50 2002
+++ easysoap-orig/src/abyss/src/threadpool.h        Fri Sep  7 14:20:34 2001
@@ -27,7 +27,7 @@

 #ifndef INTERFACE_ONLY

-#define MAX_THREAD (MAX_CONN+10)
+#define MAX_THREAD (MAX_CONN+5)
 ////////////////////////////////////////////////////////////////////
 //class CHttpThread

@@ -85,4 +85,4 @@

 #endif

-#endif
+#endif
\ No newline at end of file
```

easysoap.patch

Appendix C

Thread limit on Linux x86 systems

In the current Linux implementation, all threads and processes are ran as tasks. Unfortunatelly there is a default limit of maximum 512 tasks per system, and each user can only spawn 256 separate processes at one time. This can be a huge inconvenience if a simulation is to run more then 250 agents on one machine.

In order to overcome this, kernels prior to version 2.3.x need to be patched. There are also some modifications that need to be done do the `glibc` library [Thomason, 2001, lin].

C.1 Thread limit on pre 2.3.x kernels

For kernels earlier than 2.3.x, edit `/usr/src/linux/include/linux/tasks.h`, and modify the `NR_TASKS` value and then rebuild and install the kernel. The default value is 512. On systems without APM it can be increased up to 4090.

C.2 Thread limit on other kernels

All other kernel versions (i.e. later than 2.3.x) do not have a thread limit built in, so you do not have to modify them. However the `glibc` constraints still apply.

C.3 Modifying glibc

All the thread allocation is done via the pthread library which is part of the glibc library. Glibc has an implicit thread limit of 1024 threads per process. In order to fix these, the following steps need to be followed [Thomason, 2001]:

- get the sources for glibc and install/unpack them
- in glibc/linuxthreads/internals.h, change the size of the thread stack reserve from 2 megabytes down to 256 kilobytes with a page size of 4,096 bytes:
  STACK_SIZE (2 * 1024 * 1024) → (64 * PAGE_SIZE)
- in glibc/linuxthreads/sysdeps/unix/sysv/linux/bits/local_lim.h, change the Posix thread implementation limit from 1,024 per process to 8,192 per process:
  PTHREAD_THREADS_MAX 1024 → 8192
- rebuild and re-install the library

Appendix D

Using the code on floppy-disk

The entire code tree for the project is provided as a `tar.gz` archive on the floppy-disk. In order to unpack the code first copy the code to the harddrive and unpack it:

```
cp /mnt/floppy/dalt.tar.gz ~
cd ~; tar -xzvf dalt.tar.gz
```

In order to be able to run the program, several external libraries are required. Check the `README` file in the root of the source code tree for details on obtaining the aditional libraries and running the sample simulation provided.

Bibliography

Linux kernel tuning. web. URL linuxperf.nl.linux.org/general/kerneltuning.html. C

R. Brooks. Artificial life and real robots. In *European Conference on Artificial Life*, pages 3–10, 1992. URL citeseer.nj.nec.com/brooks92artificial.html. 1.6.2.1

Frank Cohen. Myths and misunderstandings surrounding soap. web, September 2001. URL www-106.ibm.com/developerworks/library/ws-spmyths.html. 8

Rosaria Conte, Nigel Gilbert, and Jaime Simão Sichman. Mas and social simulation: A suitable commitment. In Rosaria Conte Jaime S. Sichman and Nigel Gilbert, editors, *Multi-Agent Systems and Agent-Based Simulation*, volume 1534 of *Lecture Notes in Artificial Intelligence*, pages 1–9. Springer-Verlag, 1998. 1.2

Jim Doran and Nigel Gilbert. Simulating societies. In Jim Doran and Nigel Gilbert, editors, *Simulating Societies. The computer simulation of social phenomena*, chapter 1, pages 12–14. UCL Press, 1994. 1.2

Jim Doran, Mike Palmer, Nigel Gilbert, and Paul Mellars. The eos project: modelling upper palæolithic social change. In Jim Doran and Nigel Gilbert, editors, *Simulating Societies. The computer simulation of social phenomena*, chapter 9, pages 195–221. UCL Press, 1994. 1.2, 2

Bob DuCharme. *XML : The Annotated Specification*. Prentice Hall PTR, 1999. 1.6.1.4.1, 1.6.1.4.2

Bruce Eckel. *Thinking In C++*, volume 1. Prentice Hall, second edition, 2000. 2.1.1.1

Jacques Ferber. *Multi-Agent Systems*. Addison Wesley, 1999. 1.6.2.1

Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley, second edition, 2000. 13

Eleftherios Gkioulekas. Developing software with gnu. (incomplete), 1999. 2.1.1.2

Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000. 13

Stefan Handel, Florian Fuchs, and Paul Levi. Distributed negotiation-based task planning for a flexible manufacturing environment. Number 1069 in Lecture Notes in Artificial Intelligence, pages 183–188. Springer-Verlag, 1996. 2

B. Hayes-Roth. A blackboard architecture for control. In *Artificial intelligence*, volume 26, pages 251–321. 1985. 2.3.1.2.2

Herman Kopetz. Scheduling. In Sape Mullender, editor, *Distributed Systems*, chapter 18. Addison-Wesley, second edition, 1994. 2.3.2.2.3

Alex Lancaster. Swarm critics. web. URL `alife.org`. 3

Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1994. 1.6.1

Jan Egil Refsnes. web. URL `www.xml101.com/dtd`. 1.6.1.4.1

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 1995. 1.6.2.1

Scot Seely. *SOAP Cross Platform Web Service Development Using XML*. Prentice Hall PTR, 2002. 1.6.1.4.1, 1.6.1.4.3, A.1

David Servat, Edith Perrier, Jean-Pierre Treuil, and Alexis Drogul. When agents emerge from agents: Introducing multi-scale vierpoints in multi-agent simulations. In Jaime S. Sichman, Rosaria Conte, and Nigel Gilbert, editors, *Multi-Agent Systems and Agent-Based Simulation*, volume 1534 of *Lecture Notes in Artificial Intelligence*, pages 183–198. Springer-Verlag, 1998. 1.2, 4.6

Yoav Shoham. Multi-agent research in the knobotics. In Cristiano Castelfranchi and Eric Werner, editors, *Artificial Social Systems*, number 830 in Lecture Notes in Artificial Intelligence, pages 271–277. Spriner Verlag, 1994. 2.3.1.2.1

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997. 2.1.1.1

SUN. Java rmi. web. URL java.sun.com/j2se/1.3/docs/guide/rmi. 1.6.1.2

Andrew S. Tanenbaum. *Structured Computer Organisation*. Prentice-Hall Inc., fourth edition, 1999. 2.2.1, 2.2.1.1

Scott Thomason. Industrial linux. web, 2001. URL industrial-linux.org/ilg/performance.html. C, C.3

Mario Tokoro. Agents: Towards a society in which humans and computers cohabitate. Number 1069 in Lecture Notes in Artificial Intelligence, pages 2–3. Springer-Verlag, 1996. 2

Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders publishing, 2000. 2.1.1.2

W3C. Soap version 1.2 part 1: Messaging framework. web, December 2001. URL www.w3.org/TR/soap12-part1/. 1.6.1.4, A.1

Eric Werner. What ants cannot do. Number 1069 in Lecture Notes in Artificial Intelligence, pages 19–39. Springer-Verlag, 1996. 2, 1.6.2.1