The Zeus Agent Building Toolkit

ZEUS Methodology Documentation Part IV

The Runtime Guide

Jaron Collis, jaron@info.bt.co.uk

Divine Ndumu, ndumudt@info.bt.co.uk

Intelligent Systems Research Group, BT Labs

Release 1.01, Novemeber 1999

Index

Document History

Version 1.01 - Updated section 2 to reflect changes in batch file names

1 INTRODUCTION

This document describes how to run the agents created using the ZEUS toolkit. Whilst actually using the agents will depend on the application, all ZEUS agents are launched using the same process and all can be visualised using the application independent visualisation system.

Visualising overall system behaviour in such systems with distributed control, data and process is a notoriously difficult task. Each agent in the system has only a local view of the organisation, and the burden is on the user to integrate into a coherent whole the large amounts of scope-limited information provided by individual agents. Further, because of the complexity of multi-agent interaction and behaviour, effective visualisation assumes more importance than in single-agent systems.

Consequently the ZEUS toolkit provides a visualisation system comprising of a suite of tools that provide different perspectives of the application being visualised. Each tool interrogates the agents in the application, collates the returned information and presents this information to a user in an appropriate manner. This shifts the burden of inference from the user to the visualiser. The current tool-set includes:

(i) a *society tool* that shows the message interchange between agents in a society,
(ii) a *report tool* that graphs the society-wide decomposition of tasks and the execution states of the various subtasks,
(iii) an *agent viewer* for monitoring the internal states of agents,
(iv) a *control tool* for remotely modifying the internal states of individual agents, and
(v) a *statistics tool* for collating individual agent and society-wide statistics.

In addition to functioning in *online* mode (i.e. visualising the 'live' interactions in a multi-agent set-up), the society, report and statistics tools also support *off-line*, video-style replay of saved multi-agent interaction sessions. This multi-perspective visualisation approach gives users the flexibility to select which tool to use, when to use a particular tool, and what to visualise.

But before we consider the features of the Visualiser, the process by which agents are launched will be described.

2 RUNNING THE APPLICATION

Once the agents and any task bodies, external resources and external programs have been implemented and compiled, the next stage is to distribute the agents to their host machines and launch them.

How to Distribute Agents

Although agents can be associated with different hosts during the Code Generation phase, the Code Generator writes all source code into a local directory, (although future versions may provide support for automatic distribution to the different host machines). Hence if the application is to involve multiple hosts the developer will need to copy the source code that implements each agent and its abilities to the appropriate machine before the application can be run.

If you are distributing agents you must ensure that the destination machine has all the necessary code either copied to a directory on its local file system, or accessible through a networked drive. The following files are required for each agent:

- the agent's source code implementation, called <*agentname*>.java
- any tasks the agent is capable of performing (these were specified during the **DEF-2** activity), task name are in the format: <*taskname*>.java
- if the agent has an external resource, copy the file that implements it; you will have already specified this file's name during activity **TACF-1**)
- if the agent has an external program, copy the file that implements it; you will have already specified this file's name during activity **TACF-1**)

In addition, each host machine must have a copy of the ZEUS toolkit installed, or have access to the toolkit files via a network file system. If you are using a PC, you may need to copy across and edit some of the launch scripts - these are described in the next section.

Another prerequisite is that each machine hosting an agent must have a working Java Runtime Environment (JRE) that is version 1.2 or greater. The presence of a JRE can be easily tested by typing 'java -version' into your machine's command line, if it is installed correctly you will see a message reporting the version of the JRE installed on that machine. Otherwise you will need to install the JRE (or make it accessible) before proceeding.

You should also ensure that the JRE's CLASSPATH environment variable contains a reference to the directory that contains the ZEUS toolkit's class files. The state of this variable can be seen by typing SETENV on UNIX machines, or SET on PCs.

How to Launch Agents

The easiest way to start the application is by using the pre-written launch scripts that are created by the Code Generator. Slightly different scripts are created for Unix and PC platforms, these are:

Script Name	Agents Launched
Unix: **run1** PC: **run1.bat**	The root Agent Name Server
Unix: **run2** PC: **run2.bat**	All task agents
Unix: **run3** PC: **run3.bat**	Every other utility agent, i.e. all non-root Name Servers, Facilitators, Visualisers and Database Proxies

Differences Between UNIX and MS-DOS Scripts

There are two main differences between the scripts generated for UNIX platforms and the MS-DOS scripts generated for PCs. The most significant difference is that the UNIX scripts use the `rsh` (remote shell) command to launch agents on remote hosts using the local command line. This enables agents to be launched by a script command regardless of the host they will be running on, (although obviously whoever runs the script must have telnet permissions for the remote hosts).

Unfortunately `rsh`-like behaviour is difficult to achieve on Windows PCs, due to their lack of an operating system telnet service. (Although 3rd party software does exist that will supply telnet functionality, the basic PC scripts will have to be edited to take advantage of it). Consequently, agents on PCs will have to be launched from a local command line, this may require copying the `run3.bat` script to each PC and editing it so that only references to the agents on that machine remain.

The other difference between UNIX and MS-DOS scripts is syntactic: MS-DOS script command begin with 'start /b', whereas UNIX commands do not, instead they have an ampersand at the end of each line. The effect is the same, to start the agent as a background process (i.e. in an independent thread). Some MS-DOS script commands may use the /min flag to immediately minimise the agents' console windows on start-up to prevent the screen becoming cluttered.

How to Change an Agent's Runtime Parameters

The contents of the scripts are set during the Utility Agent and Task Agent Configuration stages, (as described in sections 4 and 5). However it is possible to edit the scripts after they have been created in order to change various aspects. The commands that launch the agents, and the flags that can be changed are listed next.

Agent Name Server Command Line Options

Agent Name Servers are launched with following command:

```
java zeus.agents.ANServer <ANS name> <flags>
```

Where the following flags are used:

-s *<dns_file>*	Specifies the path and name of the file that contains the I.P address of the root name server, (this is explained in activity **UTIL-1**)
-t *time_grain*	an integer value setting the society's time-grain, only the value set by the root name server will be used, (time grains are explained in Section 3)
-f *<file>*	Specifies the path and filename into which the root server will write its location, (this is explained in activity **UTIL-1**)
-gui	Creates a display user interface for this agent
-debug	Sets the verbosity of the agent's output stream messages
-h	Displays the usage syntax
-v	Displays the agent's version number

Facilitator Command Line Options

Agent Name Servers are launched with following command:

```
java zeus.agents.Facilitator <agent name> <flags>
```

Where the following flags are used:

-o *<ontology>*	Specifies the path and name of the file containing the application ontology
-t *cycle_time*	Sets the interval (in time-grains) between queries, (see **UTIL-2**)
-s *<file>*	Specifies the path and name of a file containing a name server location
-gui	Creates a display user interface for this agent
-debug	Sets the verbosity of the agent's output stream messages
-h	Displays the usage syntax
-v	Displays the agent's version number

Visualiser Command Line Options

Agent Name Servers are launched with following command:

```
java zeus.visualiser.Visualiser <agent name> <flags>
```

Where the following flags are used:

-o *<ontology>*	Specifies the path and name of the file containing the application ontology
-t *cycle_time*	Sets the interval (in time-grains) between queries, (see **UTIL-3**)
-s *<file>*	Specifies the path and name of a file containing a name server location
-gui	Creates a display user interface for this agent
-quick	Automatically connects the Visualiser to the specified name server and obtains the list of known agents
-debug	Sets the verbosity of the agent's output stream messages
-h	Displays the usage syntax
-v	Displays the agent's version number

Database Proxy Command Line Options

Agent Name Servers are launched with following command:

```
java zeus.agents.Facilitator <agent name> <flags>
```

Where the following flags are used:

-p *<classpath>*	Specifies the package path of the class that implements persistence, (this is described in activity **UTIL-4**)
-s *<file>*	Specifies the path and name of a file containing a name server location
-gui	Creates a display user interface for this agent
-debug	Sets the verbosity of the agent's output stream messages
-h	Displays the usage syntax
-v	Displays the agent's version number

Task Agent Command Line Options

Agent Name Servers are launched with following command:

```
java <agent name> <flags>
```

Where the following flags are used:

-o *<ontology>*	Specifies the path and name of the file containing the application ontology
-s *<file>*	Specifies the path and name of a file containing a name server location
-e <*classpath***>**	Specifies the package path of the class that connects the agent to its external program, (this is explained in activity **IMPL-4**)
-r <*classpath***>**	Specifies the package path of the class that connects the agent to its external resource, (this is explained in activity **IMPL-3**)
-gui	Creates a display user interface for this agent
-debug	Sets the verbosity of the agent's output stream messages
-h	Displays the usage syntax
-v	Displays the agent's version number

3 USING THE VISUALISER

The Visualiser is actually an agent, built from the same agent components as the other ZEUS agents. Figure 3.1 illustrates the architecture of the visualiser, which comprises a central *hub* made up of a *mailbox*, a *message handler*, a *message context database*, and a *tool launcher* window from which the different Visualisation tools can be started. This architecture follows the classic Model/View/Controller pattern, enabling the different tools to use common data in order to provide different viewpoints on the visualisation/debugging process.

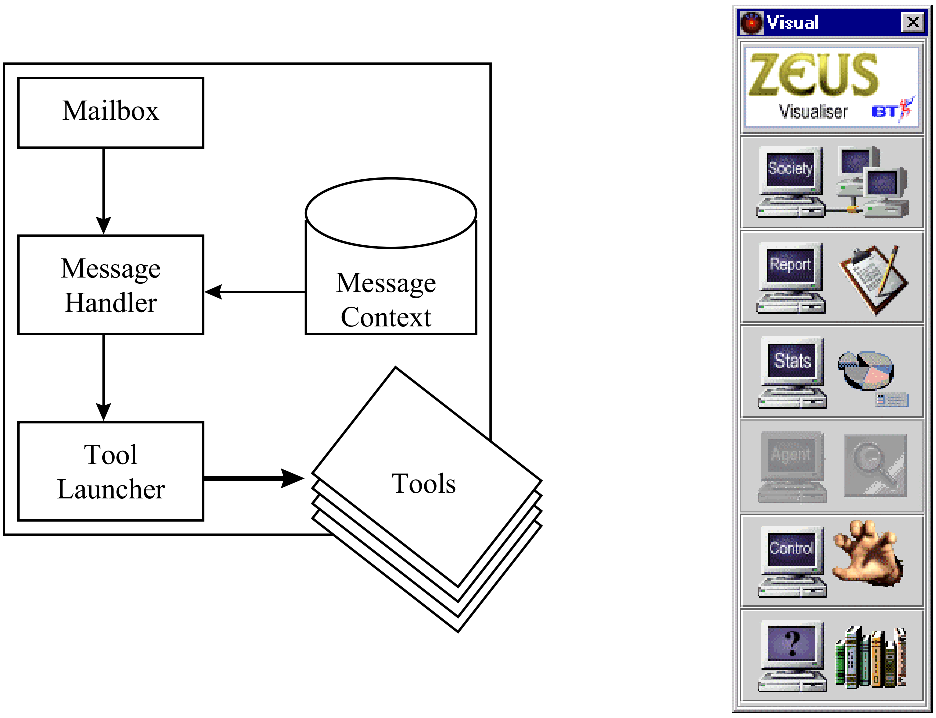

Figure 3.1: The Visualiser Architecture (left) and a screenshot (right) of the Tool Launcher Window (aka the Visualiser Hub)

The *mailbox* uses the standard ZEUS messaging mechanism to enable the visualiser to communicate with the other agents. The *message handler* processes incoming messages received by the *mailbox* and delivers them to tool instances that have registered an interest in messages of that type. Tool instances register an interest in receiving messages of a particular type by using the *message context database* that is consulted by the *message handler* during its processing. Thus, each tool instance interested in receiving report messages of a particular type from a set of agents will:

(i) use the *mailbox* to send requests to those agents that they report to the visualiser whenever events of that type occur, and

(ii) register in the *message context database* an interest in receiving all incoming report messages of the desired type.

The reply_with (outgoing message) and in_reply_to (incoming message) fields are used to associate an identifier to each message, which is unique to tool instances and event-types. This way, the message handler does not need to scan the contents of a message to determine which tool instance requires it. This arrangement allows users of the visualiser to decide at runtime the set of events they are interested in monitoring, and also to change this set at any time.

During the course of this section the features of each of the Visualiser tools will be outlined, (although operational instructions have been omitted and can be accessed using the online help facility).

3.1 The Society Viewer

The Society Viewer tool allows a user to select a set of agents and view (a) the structural organisational relationships, and (b) the messages exchanged between them. The tool will formulate its view of the society when it starts up, and can update its view whilst it is running in response to the arrival of departure of agents from the society.

The organisational relationships describe role relations such as *superior*, *subordinate*, *co-worker* or *peer*. These relationships affect how the agents might co-ordinate their activities. For example, the agents *may* be configured to try performing all tasks first, and if that fails to delegate the tasks to subordinates. The tool supports graphical layout of the agents according to role relationships. Current layout formats include a vertical tree layout emphasising the superior-subordinate relationships (as shown in Figure 3.2).

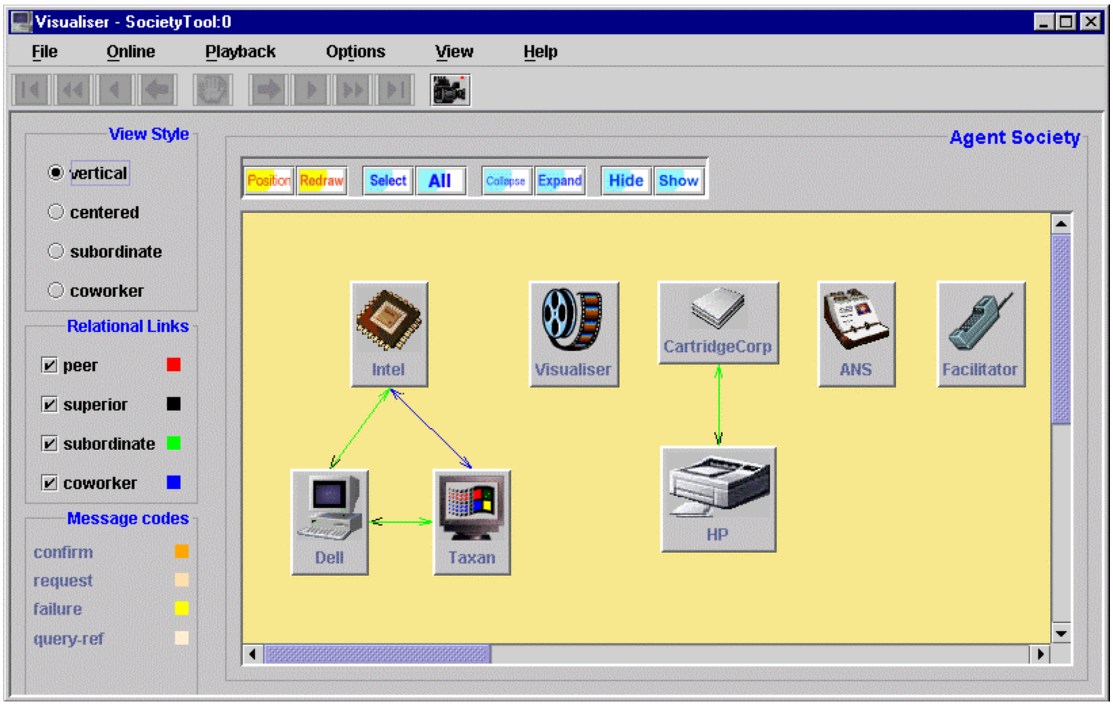

Figure 3.2: A screenshot of the society viewer tool where each icon represents an agent. The main panel will also show the exchange of messages when the agents communicate.

Also available are a horizontal layout emphasising co-workers, with groups of co-workers arranged in circles, and a horizontal layout emphasising peers, with groups of peers arranged in circles. In the layout graphs, the links between the agents are colour-coded according to their role relationship. Facilities are provided to collapse, expand, hide, show and move nodes of the various graphs.

The different layouts allow users to readily see the organisational structure of the agents and how co-ordination proceeds within that structure. This allows users to identify bugs, either in the way in which the agents are organised, or the manner in which co-ordination proceeds within the organisation. It should be noted that no single agent necessarily possesses a full picture of the organisational structure. It is the responsibility of the society tool to integrate local information returned by the agents into a global picture of the organisation.

Messages between agents are colour-coded (for easy visualisation) by type, e.g. all request messages in one colour, or by content, e.g. all messages pertaining to a particular goal in one colour. In addition, the tool supports the filtering of messages before display. Messages may be filtered by sender, receiver, type or content. For example, a filter option might state "show only counter-propose messages [type filter] from the set of agents ... [sender filter] to the set of agents ...[recipient filter] about goal ... [content filter]". These features facilitate debugging by allowing the user to focus-in on the particular messages of interest. Further, combined with the off-line video-style replay facilities with forward and backward video modes are a powerful debugging aid.

3.2 The Reports Tool

The report tool provides a global view of problem solving in a society of agents and is useful as a visualisation, debugging and an administrative tool. It allows a user to select a set of agents and request that they report to it the status of all their tasks. Next, the user can select an agent of interest and a task it owns. For the selection of agent and task, the tool generates a Gantt chart showing the decomposition of the task, the allocation of its constituent subparts to different agents in the community, and when each agent is scheduled to perform its part of the task. Other task attributes that can be shown are their costs, the priority assigned to them by the agents, and the resources they require.

The task decomposition/distribution graphs created by the report tool also show the current status of each task, i.e. either *waiting*, *running*, *completed* or *failed*. Thus, from the graph a user is immediately able to determine the overall status of a goal, and if the goal fails where exactly it did so. For easy visualisation, the different states of a task are colour-coded in the graph. Figure 3.3 shows an example of a Gantt chart generated by the report tool. The tool also provides the user facilities for collapsing/expanding sections of the graph and hiding/showing nodes on the graph – this is important in dealing with very large graphs since they allow a user to focus-in on the regions of interest.

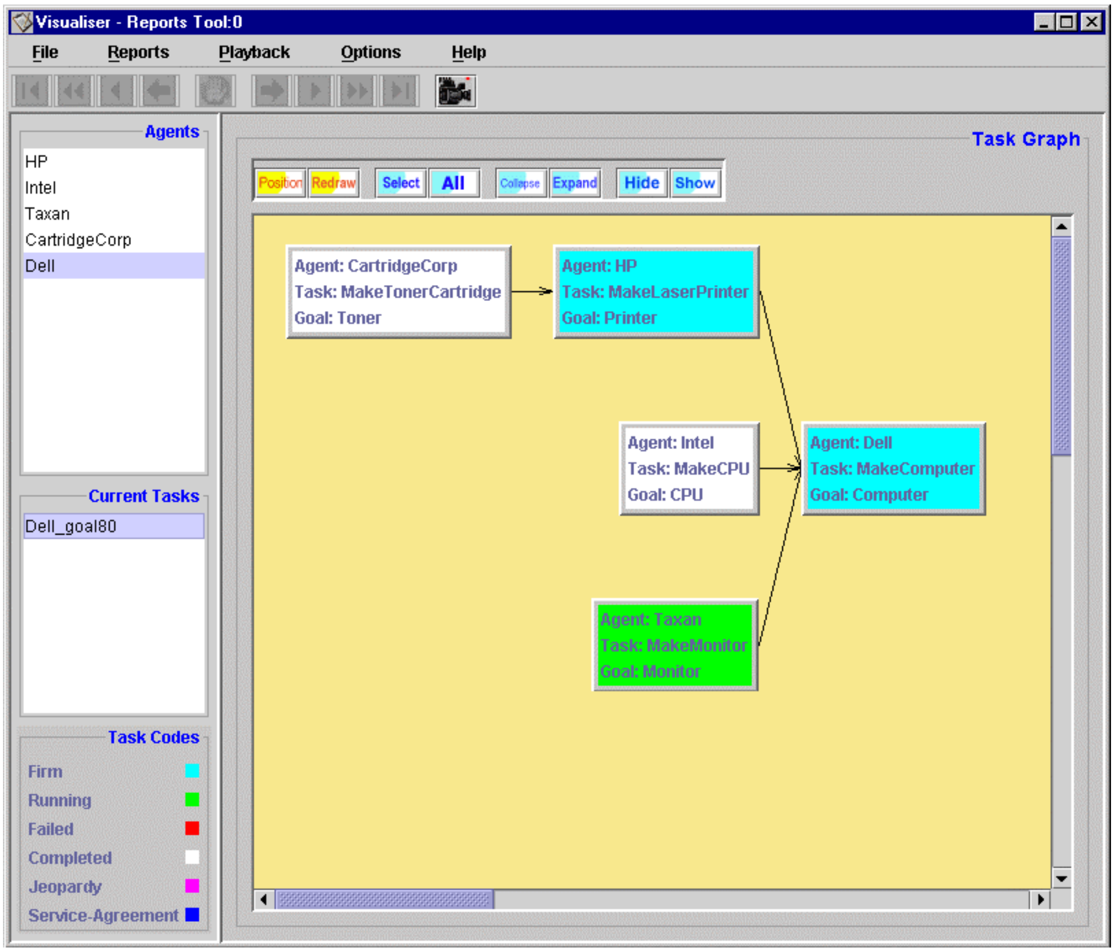

Figure 3.3: A screenshot of the Reports Tool, notice the dependencies between tasks and the colour coding according to each tasks' current state.

Just like the society tool, the report tool supports the online logging of report messages to a database and their subsequent off-line replay.

3.3 The Statistics Tool

The Statistics Tool allows a user to collate various statistics about a society of agents on a per agent basis as well as on a community basis. The statistics collected currently include: the number of messages and their types sent by the agents over a time period; the number of messages sent out by the agents in co-ordinating different goals; the average loading of the agents, i.e. the proportion of time agents spend actually executing tasks; and the co-ordination versus execution time ratio, i.e. time spent co-ordinating tasks as opposed to running them.

Statistics such as these are particularly useful in debugging or fine-tuning the performance of a society of agents. For example, using the control monitor tool and the statistics tool, a user is better able to answer questions such as "what organisational structuring and distribution of task and co-ordination know-how best minimises the time spent co-ordinating goals, and maximises the time spent executing them (i.e. increasing the profitability of the society)?". A screenshot of this tool is shown in Figure 3.4.

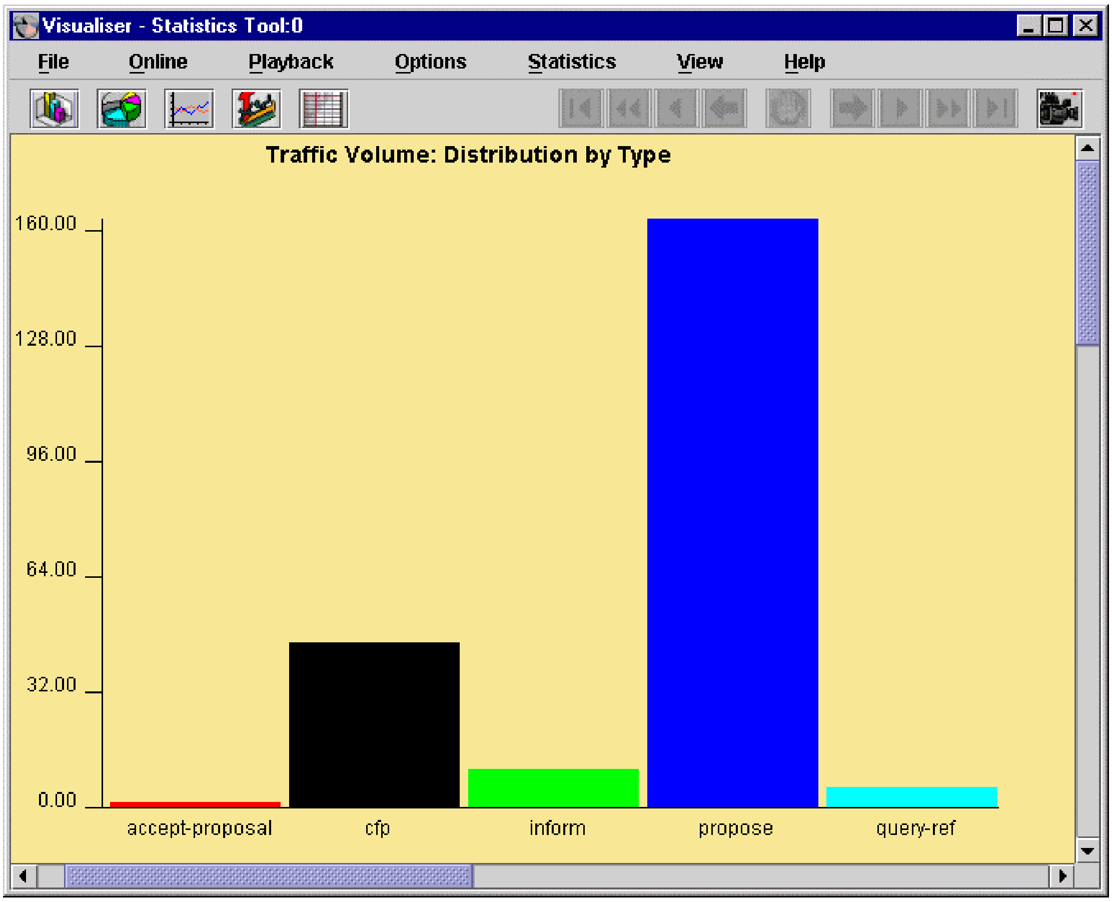

Figure 3.4: A screenshot of the Statistics Tool showing a histogram of the frequency of various message types in the current inter-agent traffic

If the agents in the society learn from experience, the statistics tool is even more useful in assessing the performance of different learning strategies. Certain statistics such as the average number of messages sent out by the agent per goal and/or the co-ordination versus execution time ratio can serve as performance metrics for measuring learning success. Like the other tools, statistics can be generated from previously saved agent sessions. Currently, the tool supports histogram, pie, line, or xy-scatter graph formats for the display of statistics. The choice of format is user-determinable, although there are pre-specified defaults.

3.4 The Agent Viewer

The Agent Viewer enables the internal state of agents to be inspected. Each agent can have its own local Agent Viewer if it is started with the following parameter:

```
-gui zeus.agentviewer.AgentViewer
```

A client-server version of this tool, which will allow the internal states of remote agents to be examined is currently being built into the Visualiser; hence the Agent Viewer option of the Visualiser Hub is not selectable.

Amongst the functionality offered by the Agent Viewer is the ability to inspect the following:

- the messages being received by the agent.
- the messages being sent out by the agent.
- a summary of the actions taken in response to incoming messages, for example, to what module of the agent the message was dispatched to for detailed processing.
- a graphical depiction of the co-ordination process of different goals by the agent. Each node of the co-ordination graph indicates a particular state of the process, and the trace of a goal on the graph summarises how it is being co-ordinated and indicates the current state of the co-ordination process.
- a list of the resources available to the agent including those allocated to the different tasks it has committed itself to.
- a summary of the results of monitoring executing tasks or tasks scheduled to start execution; e.g. this might indicate a task which failed to start because of inadequate resources or it might flag an executing task which has been stopped because it over-ran its scheduled allocation of processor time.
- a diary detailing the tasks the agent has committed itself to performing, and the current status of those tasks (i.e. waiting, running, completed or failed).

Details on how the agent components function is available in the Technical Manual, whilst some instructions on how to use the Agent Viewer are provided by the online help system.

3.5 The Control Tool

The control tool (shown in Figure 3.5) is a palette of options that allows a user to select an agent of interest and browse, add, modify or delete its current goals, resources, task specifications, organisational relations and knowledge, and co-ordination strategies. The user can also suspend or resume jobs, and also suspend, resume or even kill agents. Furthermore various agent parameters can be changed, like how the agent should split its time between performing tasks it has already committed to and co-ordinating new tasks. Thus the Control Tool reuses much of the functionality of the Agent Generator tool, allowing agents to be configured at run-time.

Figure 3.5: A screenshot of the Control Tool

The control tool is useful in debugging and/or analysing the behaviour of a society of agents by allowing users to dynamically reconfigure the society and analyse its subsequent behaviour. It provides a classic *what-if* analysis mechanism through which users can study the effects of various changes on agents' constitution, organisational relations, co-ordination behaviour, etc.

Unfortunately with the version 1.0 release of ZEUS most of the functionality of the control tool is not present, as the version that predates the Java Foundation Class still needs to be converted, although this should be available shortly.

One very useful function that is available however is the "Kill Agents" option (accessible by choosing the Bomb icon), this will send a message to selected agents telling them to cease running and close themselves down. This is by far the easiest means of terminating an agent application, especially if the agents are distributed across multiple machines.

4 THE ART OF DEBUGGING

It has already been mentioned that the debugging of distributed systems is, in general, a notoriously difficult and complex task. It is difficult to analyse and debug a single program that interacts with no other programs. It is an order of magnitude more difficult to analyse and debug distributed software that contains multiple agents. There may be *structural* or *functional* errors at the level of singular agents in the distributed set-up of agents.

Structural errors in multi-agent systems include wrong or missing acquaintance relationships between agents, missing resources, incorrectly specified (typically short) times to run tasks, etc. Naturally, we want our distributed set-up of agents to be co-ordinated and structurally and functionally 'correct'.

Functional errors refer to those concerned with the logic of the tasks that the agents are performing. However, these can also be compounded by the fact individual agents may be structurally and functionally 'correct', but the emergent behaviour of the overall set up of distributed agents may not be what was expected. These are typically due to what we refer to as *co-ordination* errors.

The suite of tools described in this paper to facilitate the debugging process when we build multi-agent systems. We call the process *debugging via corroboration* as each of the different tools provides a different and unique viewpoint to the analysis of the distributed agent application. All the tools store different and unique state data. Since no tool is capable of providing a complete picture of the entire system, the novelty of our approach comes from the fact that these different tools provide or suggest certain diagnoses. Where the evidence or suggestion of one tool corroborates the evidence from another tool, the trustworthiness in that hypothesis or prognosis is increased. There is therefore a greater likelihood that the prognosis may lead to a successful debugging of the problem. Even when the evidence from two or more tools conflict, they may eliminate or be suggestive of other possible diagnoses.

In summary, while each tool in our suite is useful as a debugging aid when used in isolation, when used in combination with the other tools they provide even greater debugging and analysis support by giving the user a multi-perspective viewpoint of the agents' behaviour. In other words, the tools used in combination when debugging offer more value that the sum of their individual contributions. The process of using the Visualiser tools to debug via corroboration can be illustrated by the following example.

4.1 Debugging via Corroboration: An Example

Consider a multi-agent society comprising seven agents A–G as shown in Figure 4.1a. The user believes the agents have been configured such that given job J1 to agent A, the decomposition of the job across the society would proceed as illustrated in Figure 4.1a, that is, J1 to agent A, J2 to agent B, J3 to agent C, etc. As no facilitator exists, all the agents are expected to have been configured with appropriate acquaintance models containing the relevant and correct relationships between agents.

Now, consider that when job J1 is given to agent A, it reports a failure in planning out the job. To debug this problem, we might use the statistics tool to review the messaging across the society pertaining to the job J1 (Figure 5b). From the figure, we note that, as expected, agent A sends out two *propose* (request) messages, presumably to agents B and C to perform jobs J2 and J3 respectively. (This can be verified via the society tool using the message filter mechanisms to review only those messages pertaining to job J1.) Also, as expected, agent B sends out one *propose* message. However, contrary to our expectations, agent C sends out two instead of three *propose* messages.

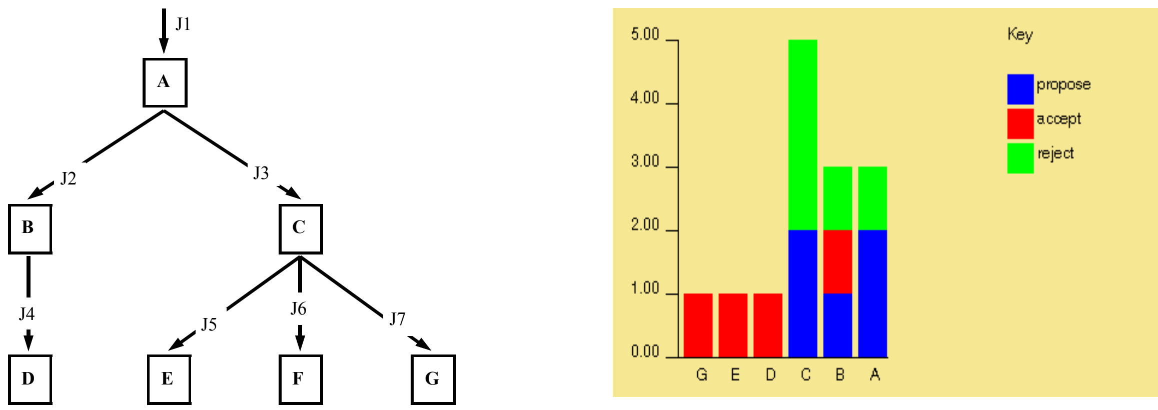

Figure 4.1: (a) Job decomposition and distribution graph for example debugging problem. The labelled boxes represent the agents in the scenario, and the labelled arrow lines show the expected job decomposition in the society. (b) The statistics tool depiction of the traffic distribution generated the goal achieve J1 given to agent A. (Note: the bar for each agent shows the messages sent out by that agent.).

We can now form the hypothesis that the cause of failure is located in agent C, probably due to one or both of:

(i) an error in the specification of the J3 task. The task specification is expected to be {J5, J6, J7}→{J3}, that is, given J5, J6, and J7 then J3 can be produced. If there is an error in the specification, for example, {J8, J6, J7}→{J3} instead, then we would get two *propose* messages sent out for jobs J6 and J7, but no equivalent *propose* message for job J8. Agent C will not send out a *propose* message for job J8 since it does not know of any other agent that can produce J8.

(ii) an error in a specification in the acquaintance database of agent C. Assuming the task is properly specified, then the most likely reason why the third propose message was not sent out would be if no agent is listed in agent C's acquaintance database capable of performing the third job.

Our principal hypothesis that the cause of failure for the overall job is located in agent C can be verified by, for example, using the micro tool to review the internals of agent C. Assume that, on doing this, it is noted from the trace of its co-ordination graph that agent C could not plan and/or co-ordinate the activities involved in job J3, thus confirming the hypothesis.

Now, regarding the two sub-hypotheses, we could use the control tool to remotely review the task specification database of agent C. Assume that, from this process, we verify that the task specification is correct. This eliminates the hypothesis of an error in the task specification, leaving us with the final hypothesis of an error in the specification of the acquaintance database of agent C. To confirm this hypothesis, we could use the control tool to directly review the acquaintance database of agent C, checking all specifications related to J5, J6 and J7 (by browsing through the acquaintance database of agent C). However, if this database is large, we could use the society tool to review the messages sent out by agent C. Assume that from this, we learn that agent C sent out messages to agents E and G to perform jobs J5 and J7 respectively. Now, we are reasonably certain that the failure to perform job J1 was due to a specification error in the acquaintance database of agent C, regarding J6 (the goal) and F (the agent). Our final hypothesis can now be confirmed, and the error corrected by using the control tool to review and modify the acquaintance database of agent C.

Now, assume the goal to achieve J1 is again presented to agent A, and this time, it reports success in distributing the task across the society (confirmed by the report tool generating a goal decomposition graph equivalent to Figure 5a, but in the format of Figure 3). Assume further, that following the task distribution, agent A later reports a failure due to inadequate resources when it tried to execute its subpart of the job.

In a rerun of the system to debug this new problem, using the report tool, we note that following successful problem decomposition across the society, agents E, F, G and C successfully started and ran to completion their respective subparts of the job. However, agent D failed to achieve J4, which in turn caused the failure of agent B to achieve J2, and in turn that of agent A to achieve J1. Now, assume the report tool showed the start of execution of the task to achieve J4, followed by a failure for some reason. Possible candidate hypotheses for this problem might include:

(i) that the J4 task had insufficient time to execute, or
(ii) that the task had sufficient time to execute but produced the wrong outputs (i.e. the task body was incorrectly specified).

To assess the two hypotheses, we could rerun the scenario and use the micro tool to monitor agent D. Assume that, from this, we notice that D successfully starts the J4 task but later stops it because the task ran out of its allocated processor time (indicated by an execution monitor message) – this essentially eliminates the second hypothesis. Now, the control monitor can be used to modify the definition of J4 to increase its estimated run-time.

Concluding Remarks

The above example illustrates just one of the aspects that might go wrong within a multi-agent system. Just as developers find the same challenges reoccurring when they build systems, (which we have begun to document in the Role Model and Realisation Guides), developers are also likely to encounter recurring runtime problems. We hope that future releases of this document will expand on this idea by presenting a library of debugging strategies that developers can use to formulate hypotheses, test them and then, if confirmed, take the appropriate remedial action. If any users would like to help in this endeavour we'd welcome their contributions.

And of course any other feedback is welcomed.
Enjoy using ZEUS!

Jaron Collis (jaron@info.bt.co.uk)