AgentScope: Multi-Agent Systems Development in Focus

Elth Ogston
Delft University of Technology
e.f.y.ogston@tudelft.nl

Frances Brazier
Delft University of Technology
f.m.t.brazier@tudelft.nl

ABSTRACT

Multi-agent systems form the basis of many innovative large-scale distributed applications. The development of such applications requires a careful balance of a wide range of concerns: a detailed understanding of the behaviour of the abstract algorithms being employed, a knowledge of the effects and costs of operating in a distributed environment, and an expertise in the performance requirements of the application itself. Experimental work plays a key role in the process of designing such systems. This paper examines the multi-agent systems development cycle from a distributed systems perspective. A survey of recent experimental studies finds that a large proportion of work on the design of multi-agent systems is focused on the analytical and simulation phases of development. This paper advocates an alternative more comprehensive development cycle, which extends from theoretical studies to simulations, emulations, demonstrators and finally staged deployment. AgentScope, a tool that supports the experimental stages of multi-agents systems development and facilitates long-term dispersed research efforts, is introduced. AgentScope consists of a small set of interfaces on which experimental work can be built independently of a particular type of platform. The aim is to make not only agent code but also experimental scenarios, and metrics reusable, both between projects and over simulation, emulation and demonstration platforms. An example gossip-based sampling experiment demonstrates reusability, showing the ease with which an experiment can be defined, modified into a comparison study, and ported between a simulator and an actual agent-operating system.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*

General Terms

Experimentation

Keywords

Multi-Agent Systems Development

1. INTRODUCTION

Agents, unlike passive nodes in traditional computer systems, analyse and react to their surroundings, autonomously making decisions and adapting to their environment. These properties pose a unique challenge in the design of distributed systems. Such knowledge intensive activities are predicated on the availability of information. As systems scale up and become more decentralised the costs of gathering information becomes an important consideration in an agent's design. Increases in complexity which are beneficial in theory may not prove cost effective in application. Or agent algorithms may set requirements that are beyond the resources of an underlying computing system.

For agent algorithms in which information obtained from the environment in a large-scale distributed system is crucial, design is not limited to the theoretical or conceptual stage of development, but is a continuous process that spans into an experimental implementation, testing and comparison process. Procedures that provide information services need to be carefully matched, and perhaps customised. The performance of an algorithm must be examined in combination with the services it uses. Possible designs must be compared in the intended application setting, and improvements in effectiveness weighed against increased cost.

This process of designing practical multi-agent systems can be viewed as an incremental development cycle, moving from theoretical studies to experimental simulations, emulations, and demonstrators, and finally to staged deployment. Experimental work plays an important role in the process. Simulation, emulation and demonstration experiments allow key aspects of system behaviour to be examined in detail in a controlled environment. Correctness and performance of algorithm implementations can be confirmed, and alternative algorithms can be carefully compared.

In contrast to this comprehensive view of the development cycle, a survey of recent papers finds that experimental work on multi-agent systems is heavily weighted towards the simulation stage. A large proportion of works appear to be strongly influenced by methodologies that view system design as occurring primarily in the theoretical or analytical stage of development.

AgentScope is a tool that supports the experimental stages of multi-agents systems development and facilitates long-term dispersed research efforts. AgentScope defines a set of generic interfaces for (1) networked communication between agents, (2) measurement and analysis of agent behaviour, and (3) the setup of experimental scenarios. The AgentScope interfaces allow protocols and experiments written for one type of platform, for instance a simulation environment, to be easily ported to other types of platforms, such as emulation environments, as research progresses through the development cycle. Real agent environments, such as the AgentScape middleware platform [15], can also be used, simplifying the transition between the experimental and deployment stages of the development cycle. AgentScope further enables the publishing of experiments, making it easier to compare algorithms with previous work, and reuse experimental scenarios and metrics.

The aim of the AgentScope project is to allow researchers to view experimental work from a more ambitious perspective than

Cite as: AgentScope: Multi-Agent Systems Development in Focus, E. Ogston and F. Brazier, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Tumer, Yolum, Sonenberg and Stone (eds.), May, 2–6, 2011, Taipei, Taiwan, pp. 389-396.

the testing of hypotheses for a single publication or project. AgentScope views experiments not as a one-off effort by a single researcher but as long-term work by many researchers. AgentScope gives support for measurement, analysis and scenario development equal priority to support for algorithm development. AgentScope promotes the creation of code whose use is not restricted to studying only the aspects of behaviour currently of interest, but that can be carried through a series of experiments, and be used throughout the development cycle.

This paper discusses the experimental stages of multi-agent systems development (Section 2), extending the argument for a comprehensive agents development cycle put forward in [11]. It further surveys existing experimental work (Section 3), and presents the AgentScope interfaces (Section 4). An example in Sections 5 and 6 demonstrates the ease with which experiments can be developed, extended and ported between different types of platforms using AgentScope. Sections 7 and 8 present conclusions and summarise ongoing and future work.

2. MULTI-AGENT SYSTEMS DEVELOPMENT

AgentScope considers experiments not as one off efforts by a single researcher but as part of long-term work by many researchers. From this point of view an experiment is a small step in a larger application *development cycle*. In multi-agent systems complex communication processes underlie important high-level procedures such as coordination, negotiation, collaboration, and coalition formation, to name a few. The deployment of such procedures in applications is more complex still. Ideas developed in theory rarely translate directly to deployment. Instead an intricate development process must often be followed to bridge the gap between theory and application.

In AgentScope, rather than placing agents and their overall behaviour foremost, the core abstraction is a *protocol*; a distributed set of components that collectively provide a specific distributed service. Agents are made up of a set of protocols that support their core behaviour. This change in focus views a multi-agent system not only from an agent's perspective, but also as a distributed system. Protocols map the abstract services employed by an agent to concrete implementations of distributed algorithms.

The development path between abstract agent design and full scale application deployment is divided into phases, a simulation stage, an emulation stage and a demonstration stage. Experiments in each stage single out the behaviour of specific protocols and test how well protocols combine to provide full agent functionality.

2.1 Protocols

A protocol supports a distributed application by providing a basic abstract service. For instance sampling, aggregation, dissemination, resource allocation, clustering, directories, and search are common distributed tasks that can be viewed as services used by higher-level applications. While each of these tasks is a relatively simple concept, the many factors that must be considered in a distributed environment often result in intricate communication and coordination patterns. Implementations require careful attention to detail and testing. Separating such tasks out simplifies testing, improves code reusability, and allows implementation details to be hidden to a great extent from the application that uses them.

Protocols consist of two parts: the abstract *algorithm* that is used to achieve the desired function, and the *implementation* of that algorithm. Algorithms can be, and often are, studied separately from an implementation. However, in an application context the two parts are best studied in combination. Implementation details can have a major impact on applicability. Assumptions made about the underlying system can lead to performance under different environmental conditions varying widely from that expected. Theoretical analysis often makes use of abstract concepts that may not have straightforward manifestations: the uniformity of random samples, the existence of clustering criteria, the presence of common parameters agreed among autonomous entities, for instance. Practical implementations often contain non-deterministic or heuristic behaviour that does not lend itself well to theoretical analysis.

2.2 Experimental Stages of Development

Protocol development commonly goes through three experimental phases between theory and end application: 1. simulation, 2. emulation, 3. use in application demonstrators. Each phase focuses on testing and improving a different aspect of a protocol implementation. Simulations test the *functionality* of a protocol to show that the basic algorithmic design is correct and complete. Emulations test functionality and performance in a *distributed setting*. Demonstrators test if the performance characteristics of a protocol, and assumptions about the system it relies on, *match the application* or class of applications in which it will eventually be used. The main practical difference between the phases is the degree to which aspects of the eventual application environment are replaced by abstract or simplified models.

Simulations involve building up a detailed understanding of an algorithm's basic functionality. The aim is often to confirm or enhance a theoretical analysis. Abstract models of the eventual application environment are used to focus in on key theoretical behaviours. Full protocol details are often not of interest. Since algorithms are usually previously untested, detailed debugging can be involved. Simulations are therefore best suited to environments that run on a single machine, using a simplified model of the eventual distributed setting. This allows for recording and analysing large amounts of data, such as the internal state of all agents, and for stopping the experiment clock to take a single synchronised snapshot of the system to test if global invariants are upheld.

Emulations are characterised by the use of a real distributed environment to confirm protocol functionality when actual communication characteristics are taken into account. Emulations are concerned with the implications of parallelisation. Simulations often take into account the location and replication of data and what messages need to be passed between agents. Emulations further consider timing and synchronisation, bandwidth requirements, the effect of communication latency and errors, and so forth. While these concerns can be partly tested in simulations, the use of an actual network is often simpler than the effort required to model it. The use of a network analogous to that used by the intended end application avoids the need to identify and model every aspect that may be of importance.

Demonstrators take a step towards testing an algorithm in a real application setting. Simulations and emulations usually focus on the full range of algorithm behaviour tested on abstract data sets. Demonstrators add models of intended uses for large-scale distributed systems based on expert analysis and real data. They test if an algorithm is suited for a specific purpose as opposed to testing its generic behaviour.

3. RELATED WORK

Experimental work is an integral part of many Agents research projects. In order to obtain a rough picture of the experimental tools and methods used in recent work a survey was made of the 61 papers presenting original work published in the Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) in 2009 and 2010 [1]. JAAMAS covers a wide range of topics, 37 of the pa-

pers surveyed presented experimental results, of these 18 ran experiments on distributed, potentially large-scale multi-agent systems. A further examination is made of these 18 papers as they represent the type of experimental work directly targeted by AgentScope.

The vast majority of work falls in the simulation phase of the development cycle. None of the 18 papers examined deployed applications, only two were based on deployed demonstrators. Of the remaining 16 all ran simulations. None of these simulations could be characterised as emulations, running on a real network. Though six papers ran more advanced simulations that modelled messages, only two of these measured communications costs and only one of those two stated that it was run in real time rather than being rounds based. Only three of the simulations used a scenario based on measurements of actual systems.

It is surprising that so little experimental work falls within the later stages of the development cycle, especially given that JAAMAS papers generally represent the more mature projects within the field. Given the importance of considering application characteristics it is disappointing that there is little work representing deployed applications, or scenarios based on expert analysis and measurements of existing systems. This imbalance may reflect the influence of development methodologies that focus on the theoretical side of Agents research, considering the bulk of important design decisions to occur in the early analytical stages of development. Methodologies, such as Gaia [16], which place an abstraction barrier between design and implementation, make an implicit assumption that implementation and application details will have little effect on system design.

Reusability is a key factor in supporting the wider spread adoption of a more comprehensive development methodology. The reuse of agent components, as discussed in [2], lowers the cost of extending previous work and running comparisons studies. We found little evidence of the reuse of agent components in the 18 works surveyed. While all the papers discussed related algorithms, only half did experimental comparisons. Four of these mentioned either needing to re-implement previous work or not being able to do a further comparison because of a lack of code.

Standard experimental platforms further improve reusability. Examining platform reuse found that of the 18 selected papers, four used existing published simulators, and two used previously published demonstrators. For five papers a custom simulator was developed, seven papers did not state what simulator was used. A good variety of simulators are in fact available [4, 6, 10, 13]. The JamesII project [5] shows a common interface can be used to allow simulations to be ported between simulators. AgentScope follows this approach, extending reusability by emphasising the desirability of porting experiments between different types of platforms [14] rather than focusing on simulations.

A third aspect of reusability concerns the development of common scenarios, metrics and experiments. In the papers surveyed, scenario and metric reuse was more common than direct reuse of code: nine papers were based on previously developed scenarios, and three extended previous scenarios. Twelve papers used previously published metrics, and one used an extension of a previous metric. Six papers developed custom scenarios, five of these created custom metrics. JamesII [5] includes mechanisms to reuse experiments. In AgentScope the interfaces for implementing metrics and building experimental scenarios are given equal importance to the interface on which agents are built. The intention is to encourage the creation of standard testbeds, which make input from domain experts, and data sets more accessible. The success of agent competitions gives an indication of the value of this approach. Three of the four simulators that were explicitly reused in the papers surveyed were from agents competitions: Robocup [9], Robocup Rescue [7], and the Trading Agents Competition [8]. It is perhaps significant that these simulators are associated with full test beds, including scenarios and metrics, and for which code for previous solutions is available. AgentScope aims to improve the extendability of this approach by enabling the creation of more generalised scenario libraries.

4. THE AGENTSCOPE INTERFACES

AgentScope gives support for measurement, analysis and scenario development equal priority to support for algorithm development. It defines three interfaces that separate out the core behaviours needed to implement distributed protocols and run experiments: (1) Networked Communication (2) Measurement and Analysis and (3) Experimental Control. The Protocol Network Interface provides the methods and classes needed for agent protocol instances to communicate with each other. Protocol code written on the Protocol Network Interface can be designed to be used throughout the whole development cycle. The Measurement and Analysis Interface defines a generic method of recording protocol behaviour during an experiment. The specific measurements of interest generally change between experiments. The Measurement and Analysis Interface is designed to allow measurement code for a protocol to be easily swapped out and replaced depending on the experiment being run. The Experimental Control interface defines a generic method of setting up experiments so that an experimental scenario can be easily reused, or ported between platforms.

AgentScope promotes the creation of code whose use is not restricted to studying only the aspects of behaviour currently of interest, but that can be carried through a series of experiments, and be used throughout the development cycle. AgentScope interfaces are intentionally minimal. The restrictive interfaces encourage developers to use methods that are as generic as possible to the many environments encountered during the development cycle.

AgentScope interfaces are two sided. On one side they provide a generic set of classes and methods for use by protocol developers. On the other side they provide a set of classes and methods that a *backend* must implement in order to run AgentScope based protocols and experiments. In a backend, a set of "adaptor" classes map a platform's functionality to that required by AgentScope. These adaptor classes can provide additional methods that allow experiments to access functionality that is specific to a particular platform. In the following sections we give details of the protocol developer side of each interface. In Section 6 we discuss backend adaptors. The AgentScope interfaces are implemented in Java.

4.1 Protocol Network Interface

The Protocol Network Interface is a stand alone package defining the basic classes and interfaces that are needed to write protocols. The core abstraction is a "protocol": an implementation of a distributed algorithm that provides a particular service to "agents" located on nodes in a distributed system. Each participating agent runs an instance of the protocol, and collectively these protocol instances provide the service to their agents. For instance a directory protocol might maintain a list of active agents, an aggregation protocol might calculate the sum of a given dynamic variable or a load balancing protocol might distribute a set of tasks between agents.

Protocol instances each have an individual "address", which allows them to communicate with each other by exchanging "messages". They are triggered into action on receipt of a message, in response to "events" that are set to occur at a particular time (according to a local "clock"), or in response to a request made by their agent. Protocols also have "names" to allow particular in-

Table 1: The Protocol Network Interface

METHODS OF PROTOCOL CLASS
Protocol(Agent myAgent, String name)
sendMessage(Message m)
receiveMessage(Message m)
scheduleEventAt(String note, long time)
scheduleEventWithDelay(String name, long timeFromNow)
triggerEvent(Event p)
getAddress() returns Address
getName() returns String
getClock() returns Clock
start()
METHODS OF MESSAGE CLASS
Message(Address to, Address from)
to() returns Address
from() returns Address
METHODS OF EVENT CLASS
Event(Address addr, String name, long time)
getName() returns String
getAddress() returns Address
getTime() returns long
METHODS OF AGENT INTERFACE
addProtocol(Protocol p, String name) returns Address
getProtocol(String name) returns Protocol
sendMessage(Message m)
scheduleEvent(Event p)
getClock() returns Clock
METHODS OF ADDRESS INTERFACE
sameAs(Address a) returns boolean
sameAgent(Address a) returns boolean
METHODS OF CLOCK INTERFACE
currentTime() returns long
unitsPerSecond() returns double

stances of a given protocol to be located when the address of an agent is known, but not the specific address of the named protocol on that agent.

The Protocol Network Interface consists of three classes - Protocol, Message and Event - and three interfaces - Agent, Address and Clock (Table 1). The classes represent the main objects that a protocol manipulates, the interfaces represent supporting concepts from the system the protocol exists within. A protocol implementation is written by subclassing the Protocol class. Subclasses of Message and Event are used to represent protocol specific messages and events. The Agent interface provides methods for sending and receiving messages and setting and receiving events.

4.2 Measurement and Analysis Interface

The Measurement and Analysis Interface provides classes in which to define: (1) the data that should be recorded about a protocol's behaviour during an experimental run, and (2) how that data should be manipulated to provide the final output of the experiment. The core abstraction is a "logbook" - essentially a blank space in which a protocol instance records information, along with methods that define how to process that information. Logbooks are located on Agents. Each protocol instance can have one or more logbook instances associated with it.

The Measurement and Analysis Interface consists of two classes, LogBook and its subclass ActiveLogBook, and one interface, Logger (Table 2). A LogBook stores and manipulates the raw data recorded during an experiment. A Logger interacts with the backend to aggregate the recorded data into a final single location. In order to write an experiment a designer must subclass LogBook to specify the data of interest, and methods for analysing it.

There are two modes in which a logbook can operate - passive and active. When a logbook is passive, the protocol code specifies the data to record. When a logbook is active the logbook code

Table 2: The Measurement and Analysis Interface

METHODS OF LOGBOOK CLASS
LogBook(String name)
aggregate(LogBook moreData)
clear() returns LogBook
writeData(File directory)
combineData(File[] inputDirs, File outputDir)
getName() returns String
METHODS OF ACTIVELOGBOOK CLASS
ActiveLogBook(Protocol p, String name)
recordProtocolState()
METHODS OF LOGGER INTERFACE
addLogBook(LogBook l)

specifies the data to record, a protocol need not contain specific logging code or know of the log's existence. Active logging is better suited when fully experiment-generic protocol code is desired. Passive logging is better suited to detailed measurements that involve recording events as they occur.

The more basic class, LogBook operates in passive mode. In passive mode, a protocol instance must be informed of the a logbook's existence, for example by the logbook instance registering as a listener to the protocol instance. The protocol code specifies what information to store. ActiveLogBook subclasses LogBook to add functionality for recording data independently of a protocol implementation. In active mode a logbook instance holds a pointer to a protocol instance. The data to recorded is specified within the logbook code. When triggered, for instance by a periodic signal from the logger, the logbook calls methods on the protocol to extract data.

Logbooks store data in a distributed manner within the agents which they are monitoring. A backend adaptor implementation of the Logger interface specifies details of how individual logbooks are routed by the experimental system to produce the final output of an experiment at a central location. The Logger combines data from the individual agent logs into a single central logbook instance. The LogBook.aggregate() method is used to specify how data from individual logs should be combined. The LogBook.writeData() method specifies how the final results for an experiment should be recorded to a specified a directory, for instance in the form of graphs of tables. The LogBook.combineData() method further specifies how the output produced by the writeData() method for several different experimental runs can be combined into a single set of results.

The AgentScope toolkit contains a supporting package of classes for storing, manipulating, and analysing data and drawing graphs and tables with which logbooks can be implemented. It also contains a set of generic logbooks, for instance for recording, analysing and graphing series or time sequences of values. Additionally, a logger implementation that is built entirely on the Protocol Network Interface is provided. In this case the logger is a protocol that uses messaging to move data, and events to trigger active log functions. For backends that contain specific logging support, such as synchronised triggers for taking system snapshots, a logger implementation can be built that makes optimal use of that support.

4.3 Experimental Control Interface

The Experimental Control Interface provides a means of organising the setup of an experiment in a flexible manner. Both the need to change experiment configuration and the need to port experiments between backends are taken into consideration. The setup and running of an experiment is divided into several parts. First, "experiments" are distinguished from "trials". An experiment is a full experimental scenario while trials are single runs of that sce-

Table 3: The Experimental Control Interface

METHODS OF ENVIRONMENT INTERFACE
runTrial(int numAgts, Initializer i, LogBook l) returns LogBook
METHODS OF EXPERIMENT INTERFACE
run(File outputDir)
runTrial(Trial t, File outputDir)
combineOutput(File[] inputDirs, File outputDir)
METHODS OF INITIALIZER CLASS
Initializer(Trial trial)
initializeAgent(Agent a, Logger l)
finalizeSetup() returns boolean
getTrial() returns Trial
METHODS OF TRIAL CLASS
Trial(String name, int numAgents, int trialLength)
getNumAgents() returns int
getName() returns String
getTrialLength() returns int

nario. Second, the "core" experimental scenario is separated from the parts of the experiment that may be configured differently in different trials.

The division of purpose is represented by the Experimental Control Interface classes and interfaces: (1) an Experiment provides the interface through which to run a series of trials, (2) an Environment captures information about the setup of the backend environment in which the scenario is to be run, (3) an Intializer stores information on the setup of agents for a core experimental scenario, and (4) a Trial stores information on the configuration of a particular trial. (Table 3)

An Experiment provides an interface for running a trial or series of trials. Within an Experiment a designer specifies the Environment or Environments to use, Initializers, a core series of trials to run, and how the output of a series of trials should be combined to give the final experimental results. Experiment provides the basic methods on which user interfaces can be built. The AgentScope toolkit provides a simple command line UI, a web-based UI and a generic file manager that provides methods for running complex series of trials and storing them to a standard directory structure.

An Environment is the part of the adaptor for a given backend with which an Experiment interacts. An Experiment runs a trial on a backend by calling that backend's Environment.runTrial() method. The runTrial() method takes an Initializer as input which defines the generic agent setup. Running an experiment on a new backend only requires a switching Environments.

An implementation of Initializer specifies how agents should be set up to run a given scenario, in a backend independent manner. At the start of an experiment a backend creates empty agents, which are then passed to the initializer. The initializer adds the required protocols and logbooks to the agents and sets up the initial connections between them.

A Trial keeps track of trial specific configuration details. An Initializer specifies the setup of a generic scenario, it can be given a Trial instance to query for parameters that may vary between trials. Trials can for example be used to vary the values of variables, the number of agents, protocol implementations used, initial connections between agents, experiment event series, logbooks used, input datasets, etc. An experiment designer can thus define the fixed and variable parts of an experimental setup through defining a core Initializer and a Trial or set of Trials specifying what may change between experiment trials.

5. SAMPLING EXPERIMENT EXAMPLE

Writing an experiment in AgentScope requires developing a protocol to be tested, logbooks to measure its performance, and an experimental setup in which to run. This process is demonstrated for the implementation and testing of a basic sampling protocol, SimpleGossip. A *sampling service* is a degenerate form of directory service that when queried returns a randomly chosen address of an agent in the system. SimpleGossip is an unsophisticated gossip-based sampling protocol. In gossip-based sampling each agent maintains a cache of items. Each item stores the address of an agent. Pairs of agents periodically "gossip" with each other, exchanging items from their caches. Repeated gossiping creates a continuous mixing procedure in which items become spread randomly throughout the agent caches. When an agent needs a random address, the sampling protocol returns a random item from its cache. Ideally the samples returned by a sampling protocol will follow a uniform random distribution. The exact gossiping procedure, as well as a protocol's response to node churn, message loss and other underlying system characteristics determine the degree to which it meets this requirement. A detailed discussion of gossip-based sampling is presented in [12].

The following sections provide the bulk of the code needed to implement and test SimpleGossip using AgentScope and some convenience classes from the AgentScope toolkit. First a basic simulation experiment is developed in Sections 5.1-5.3. In Section 5.4 this experiment is extended to a more complex setting involving node churn, and a comparison to an existing protocol in performed. In Section 6 a further version of the experiment is run on a full distributed agent middleware platform, AgentScape [15].

5.1 Protocol Development

The core SimpleGossip protocol, given in the code lines 1-45, is developed on the Protocol Network Interface. SimpleGossip subclasses protocol. Each SimpleGossip instance contains an address-item cache of size C (line 4). Periodically, with some interval t seconds (lines 34-37), each instance chooses a gossip partner and sends it g of the items from its cache (lines 13-16). This agent responds by returning g items from its own cache (lines 17-23). Notice that some time can pass between when a request is sent and the reply is received. No attempt is made to manage the ordering of gossips. Subsequent incoming gossips can be handled in this interval. Nor does SimpleGossip keep track of the requests it makes, or notice when a gossip fails and no reply is received. An agent joins a SimpleGossip protocol by filling its cache with items for its own address, then initiating a gossip with a bootstrap agent already in the protocol (lines 25-31).

```
1  public class SimpleGossip extends Protocol{
2    int t=1; int C=25; int g=3;
3    Address bootstrapAgent;
4    ItemCache theCache;
5    public SimpleGossip(Agent myNode, Address bootstrapAgent){
6      super(myNode, "SimpleGossip");
7      this. bootstrapAgent = bootstrapAgent;
8      theCache = new ItemCache();
9    }
10   public Address getRandomAddress(){
11     return theCache.getRandomItem().getAddress();
12   }
13   protected void initiateGossip(Address partner){
14     Item[] toSend = theCache.removeItems(g);
15     sendMessage(new GossipRequestMessage(partner, getAddress(), toSend));
16   }
17   protected void receiveGossipMessage(GossipMessage m){
18     theCache.addAll(m.getItems());
19     if(m.isRequest()){
20       Item[] toSend = theCache.removeItems(g);
21       sendMessage(new GossipReplyMessage(m.from(), getAddress(), toSend));
22     }
23   }
24   @Override
25   public void start(){
26     theCache.createAndAddOwnItems(C);
27     scheduleEventWithDelay(''GOSSIP'', t);
28     if (bootstrapAgent != null) {
29       initiateGossip(bootstrapAgent);
30     }
31   }
32   @Override
33   public synchronized void triggerEvent(Event p){
34     if (p.getName().compareTo(''GOSSIP'') == 0) {
35       scheduleEventWithDelay(''GOSSIP'', t);
```

```
36          initiateGossip(getRandomAddress());
37        }
38      }
39      @Override
40      public synchronized void receiveMessage(Message m) {
41        if (m instanceof GossipMessage) {
42          receiveGossipMessage( (GossipMessage) m );
43        }
44      }
45    }
```

5.2 Basic Experimental Setup

In the AgentScope Control Interface an initialiser is used to define the basic setup of agents in an experimental scenario (lines 46-61). For the sampling scenario this involves creating a SimpleGossip protocol instance for each agent, and informing each protocol of the bootstrap agent's address (line 51).

```
46  public class SamplingInitializer implements Initializer {
47    private int agentCount = 0;
48    Address bootstrapAgent = null;
49    @Override
50    public void initializeAgent(Agent a, Logger l){
51      SimpleGossip p = new SimpleGossip(a, bootstrapAgent);
52      if(bootstrapAgent == null){
53        bootstrapAgent = p.getAddress();
54      }
55      agentCount++;
56    }
57    @Override
58    public boolean finalizeSetup(){
59      return agentCount == getTrial().getNumAgents();
60    }
61  }
```

The sampling experiments are run on the Platform 9 3/4 simulator (line 65), the custom simulator used in [12]. A subclass of the Experiment class is used to define backend setup (lines 62-72). In the initial experiments specific SamplingProtocol parameters are not varied, so the the basic Trial class can be used directly.

```
62  public class SamplingExperiment extends Experiment{
63    @Override
64    public void runTrial(Trail t, File outputDir){
65      Environment env = new P934Environment();
66      env.setTrialLength(t.getLength());
67      Initializer init = new SamplingInitializer(t);
68      LogBook log = new DoubleValueLog();
69      LogBook results = env.runTrial(t.getNumAgents(), init, log);
70      results.writeData(outputDir);
71    }
72  }
```

5.3 Measurement

Measurement involves creating a set of logs to record metrics of interest to an experiment. The uniformity of the samples returned by SimpleGossip can be tested by using sampling to estimate the total number of agents, N, in the system [12]. Using the inverted birthday-paradox, let x be the total number of items seen before two items for the same agent are detected. The estimate of N is then $x^2/2$. Agents can make repeated estimates of the network size by watching the stream of incoming items produced by SimpleGossip. The statistical accuracy of this estimate gives a measure of the uniformity of the distribution from with the samples are drawn.

A listener model is used to allow objects to register to be notified each time SimpleGossip receives a new item. A class, SizeEstimate, watches the item stream and generates size estimates using the inverted birthday-paradox method. In turn SizeEstimate informs listeners registered with it each time a new estimate is generated. These estimates are recorded using the DoubleValueLog class, from the AgentScope toolkit (lines 73-113). DoubleValueLog uses the DataSet class from the AgentScope toolkit to store and analysis values (line 74). The Chart class from the AgentScope toolkit is used to produce graphs of these values (lines 92-104).

```
73  public class DoubleValueLog extends LogBook implements ValueListener<Double>{
74    DataSet values;
75    public DoubleValueLog(String name){
76      super(name);
77      values = new DataSet();
78    }
79    @Override
80    public synchronized void newValue(Double v) {
81      values.addValue(v);
82    }
83    @Override
84    public synchronized void aggregate(LogBook moreData) {
85      if (moreData instanceof DoubleValueLog) {
86        DoubleValueLog l = (DoubleValueLog) moreData;
87        values.addValues( l.getValues() );
88      }
89    }
90    @Override
91    public void writeData(File directory){
92      Chart c = values.graphDistribution(1, getName());
93      c.saveImageAndSerializedChart(directory, getName());
94    }
95    @Override
96    public void combineData(File[] dirs, File outputDirectory) {
97      LineChart outputChart =
98        new LineChart(getName(), ''Size Estimate'', ''Number of Occurrences'');
99      for(File f: dirs){
100       File nextFile = new File(f, getName() + ''.ser'');
101       LineChart c = (LineChart) Chart.readSerializedChart(nextFile);
102       outputChart.addDataSeries(f.getName(), c.getDataSeries(''dist''));
103     }
104     outputChart.saveImageAndSerializedChart(outputDirectory, getName());
105   }
106   @Override
107   public synchronized LogBook clear(){
108     DoubleValueLog l = new DoubleValueLog(getName());
109     l.values = values;
110     values = new DataSet();
111     return l;
112   }
113 }
```

In order to specify the measurements to be made during the experiment, the initializeAgent() method of SamplingInitializer (lines 50-56) is extended (lines 116-124) to create a SizeEstimate (line 118) and log (line 120) on each agent, and to register each log with the platform logger (line 122) . The logger takes care of transferring the values recorded in each agent log to a central log. To gather data the logger calls the clear() method (lines 107-112) on each registered agent log. The logger gives this data to the central log through the aggregate() method (lines 84-88). This central log is specified in Experiment.runTrial() to be another instance of DoubleValueLog (line 69).

```
115   @Override
116   public void initializeAgent(Agent a, Logger l){
117     SimpleGossip p = new SimpleGossip(a, bootstrapAgent);
118     SizeEstimate e = new SizeEstimate();
119     p.addListener(e);
120     DoubleValueLog log = new DoubleValueLog(''SizeEstimate'');
121     e.addListener(log);
122     l.addLogBook(log);
123     ...
124   }
```

Experiment.runTrial() also specifies what should be done with the central log at the end of an experiment (line 70). The DoubleValueLog write method produces a chart of the distribution of values recorded (lines 91-94).

An experiment series is defined in the run method of the SampleExperiment class (lines 126-131). Two trials are specified, one with 100 agents and one with 1000 agents. Both are run, and the output is combined using the DoubleValueLog.combine() method (lines 133-135). The FileManager class from the AgentScope toolkit is used to specify the directory structure for storing results. The FileManager runTrial() and combineData() methods simply call the corresponding Experiment methods with appropriate File arguments.

```
125   @Override
126   public void run(File outputDir) {
127     FileManager fm = new FileManager(this, outputDir);
128     fm.runTrial(new Trial( ''SimpleGossip100'', 100, trialLength) );
129     fm.runTrial(new Trial( ''SimpleGossip1k'', 1000, trialLength) );
130     fm.combineOutput();
131   }
132   @Override
133   public void combineOutput(File[] inputDirs, File outputDir) {
134     (new DoubleValueLog(''SizeEstimate'')).combineData(inputDirs, outputDir);
135   }
```

The final output of the experiment is shown in Figure 1. The figure shows the distribution of size estimates made by all agents during a run. Vertical lines mark the average. With 100 agents SimpleGossip performs reasonably well, on average estimating the

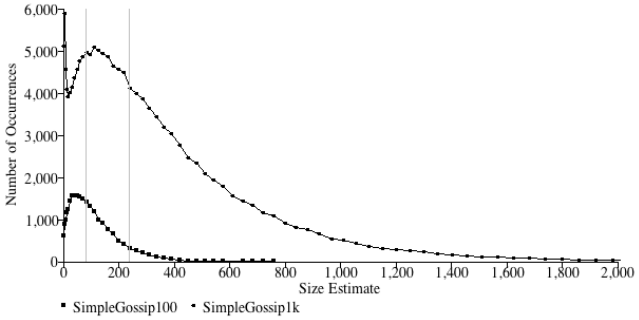

Figure 1: Size estimates: basic simulations.

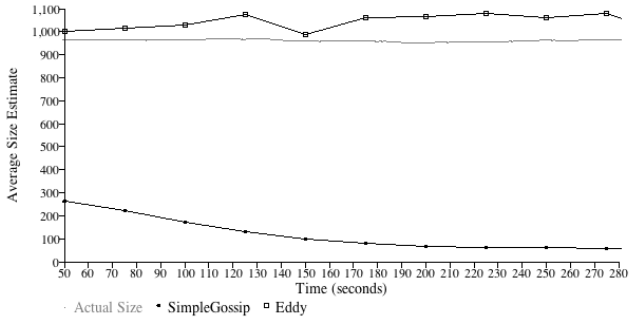

Figure 2: Size estimates: 1000 agents with churn.

system size to be 82.6. With 1000 agents however it gives an average size estimate of 238.7 showing that SimpleGossip is not a precise sampling protocol.

5.4 Extending the Experiment

The experiment described above can be easily extended. Alternative sampling protocols can be tested against SimpleGossip to directly compare performance. Additional metrics can be recorded. Or the environment in which the scenario is run can be modified.

In order to compare SimpleGossip to alternative sampling protocols a SamplingTrial class is created which specifies which protocol to use through a getSamplingProtocol() method. The SamplingInitializer calls this method when setting up an agent (line 51). A library version of the Eddy protocol, described in [12], can then be run in the scenario developed for SimpleGossip simply by creating and using a corresponding SampleTrial.

Analysing different aspects of SimpleGossip's performance only requires adding or modifying logbooks. For instance, to record how the size estimates produced by the protocol change over time, the DoubleValueLog used by the agents (line 120) can be replaced by a TimeStampedValueLog and the DoubleValueLog used by the SamplingExperiment can be replaced with a TimeStepsValueLog (line 69). Both of these are generic logs from the AgentScope toolkit. TimeStampedValueLog extends DoubleValueLog, recording the time at which each value is logged, and TimeStampedValueLog uses these times to divide values into time steps.

Finally, the environmental conditions in which SimpleGossip operates can be modified by configuring the Platform 9 3/4 backend. The simulator can be set to implement churn, causing agents to fail over time and adding new agents by calling the P934Environment.-setChurn() method after creating the environment in SamplingExperiment.runTrial() (line 65).

Figure 2 shows the end result of these changes. The figure compares the ability of SimpleGossip and Eddy to estimate system size over time in a system with churn and an average size of 1000 agents. It highlights an important failing of SimpleGossip, as the agent set changes the item set is not adapted, resulting in the quality of samples degrading over time. Eddy shows that a more complex protocol can manage the item set in a way that allows it to estimate system size fairly accurately.

6. BACKENDS

The network, measurement, and control interfaces abstract the experimental environment in which protocols are tested. In order to use an alternative platform with a experiment written on the AgentScope interfaces an experimenter need only write a small set of adaptor classes. A backend adaptor maps the abstract concepts from the AgentScope interfaces onto concrete implementations within a platform. While backends all provide the same basic functionality, they may vary significantly in the performance guarantees they provide for those functions. For instance a simulator may guarantee that messages will always be delivered with minimal delay, while in an emulation environment message and agent failures may be common occurrences. Different environments also include different support functions. Single machine environments can easily provide support for precise measurements of agent behaviour. In distributed environments measurement can require expensive coordination and communication, and thus may necessarily be less detailed. It is therefore natural that as a protocol proceeds through the development cycle the most appropriate environment for testing it will change.

In the example experiment the Platform 9 3/4 simulator can be replaced by the AgentScape agent middleware [15] simply by changing which Environment is created in line 65. Figure 3 compares the size estimate distributions for the Eddy protocol on 50 agents running in AgentScape and on Platform 9 3/4. Each backend has its advantages and disadvantages. Since AgentScape is a full agent operating system, testing is not limited to aspects that were designed into the environmental model. Run with the 1 second gossip interval, t, used in the original experiments, the Eddy protocol swamps AgentScape communications. Gossiping protocols are often designed assuming low-cost communication methods, such as UDP [3], while AgentScape communication is designed to be reliable and to maximise security. The "AgentScape Congested" trial shows that messages timing out in Eddy results in an under-estimate of the system size. Slowing the gossip rate to 10 seconds shows that Eddy performs roughly equally both in simulation and in the real environment, indicating that it does not have any inadvertent dependancies on shared data or the synchronised timing of the simulation environment. AgentScape, however, is limited to testing smaller numbers of agents than the simulator since each AgentScape agent runs as one or more threads, nor can it easily be setup to mimic churn. Run on AgentScape, the deficiencies of the SimpleGossip protocol seen in Figures 1 and 2 would not be as apparent.

7. DISCUSSION

The design of AgentScope centres on increasing the potential impact of experimental work. The example in Section 5 demonstrates the following improvements over an ad-hoc platform-specific approach to experimental design:

- *Explicit services:* Rather than viewing services as being provided by the backend platform, services (protocols) are moved above the experimental interface. This ensures that the full cost and complexity of an agent algorithm is clearly visible. It also widens the range of platforms and the range of agent

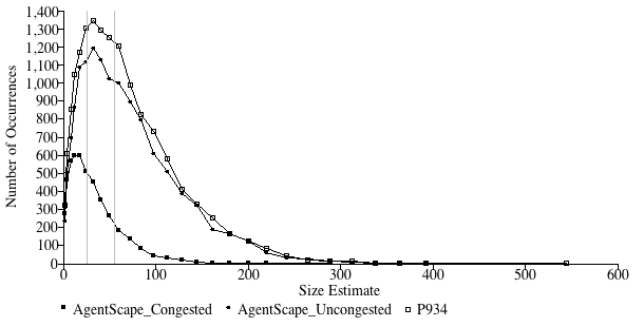

Figure 3: Size estimates: 50 agents on AgentScape.

models that can be supported. Finally, it allows AgentScope to be limited to a small simple interface that is easier to adopt and integrate with existing work.

- *Parity of protocol, scenario, and measurement:* By giving the interfaces for protocol development, experiment development and measurement equal weight, the need for reusability of all three is emphasised. An agent or protocol designer need not start from scratch, but can use existing services, scenarios and metrics. This allows researchers to focus on developing and testing the novel aspect of their ideas.

- *Abstraction of platform type:* Experiments are not specific to one type platform. The reusability of code over different stages in the development cycle is improved, and direct comparisons of performance on different platforms can be made. This has implications for the backends used, each platform need no longer provide a full solution. Timing is an important example, rather than requiring that distributed platforms include the ability to do tests with synchronised time or to create system snapshots, an approach can be used where these tests are done on a simpler centralised platform, and other metrics used to confirm that performance does not change when an experiment is run in a distributed setting.

8. CONCLUSIONS AND FUTURE WORK

This paper examines a comprehensive, long-term approach to multi-agent systems development, and in particular the experimental phases of development. A survey of recent experimental work observes a scarcity of experimental work in the later stages of development, along with a limited level of code reuse. AgentScope, a small set of interfaces on which platform generic experiments can be built is presented. The AgentScope interfaces abstract agent communication, the full setup of experimental scenarios and performance measurement. A demonstration experiment shows how AgentScope can support rapid development of flexible and reusable experiments.

AgentScope promotes a view of the multi-agent systems development process as a long-term research effort by a community of researchers, developing ideas from theory to practice. AgentScope's design centres around improving researchers' ability to transfer ideas between projects, groups, or institutions. The aim is to enable a comprehensive development process by improving the reusability of all parts of a experiment. Improvements in reusability support the creation of libraries of protocols, scenarios and metrics that can allow researchers to quickly incorporate previous work into their own experiments. With such changes, experiments become more easily reproducible, and new algorithms can be directly compared to old, making improvements and tradeoffs clear. Published libraries of scenarios, in which practitioners share their experience of expected use-cases, can be created, enabling input from industry experts to be incorporated in academic work.

Future work involves developing libraries of protocols, scenarios and metrics. Current work in this area focuses on the domain of distributed energy resource management, a highly multi-disciplinary field. The aim is to show that sophisticated agent-based experimentation can be made accessible to a broad audience. The Measurement package and Web Interface package for the AgentScope toolkit are also under development.

9. REFERENCES

[1] *Autonomous Agents and Multi-Agent Systems*, volumes 18-21. Springer, 2009-2010.

[2] P. S. da Silva and A. C. V. de Melo. Reusing models in multi-agent simulation with software components. In *AAMAS '08: Proc. 7th Int Joint Conference on Autonomous Agents and Multiagent Systems* , pages 1137–1144, 2008.

[3] N. Drost, E. Ogston, R. V. van Nieuwpoort, and H. E. Bal. ARRG: real-world gossiping. In *HPDC '07: Proc. 16th Int. Symposium on High Performance Distributed Computing*, pages 147–158, 2007.

[4] L. Gasser and K. Kakugawa. MACE3J: fast flexible distributed simulation of large, large-grain multi-agent systems. In *AAMAS '02: Proc. 1st Int. Joint Conference on Autonomous Agents and Multiagent Systems*, pages 745–752, 2002.

[5] J. Himmelspach, M. Rohl, and A. M. Uhrmacher. Component-based models and simulations for supporting valid multi-agent system simulations. *Applied Artificial Intelligence*, 24:414–442, 2010.

[6] B. Horling, R. Mailler, and V. Lesser. Farm: A Scalable Environment for Multi-Agent Development and Evaluation. In A. G. C. Lucena, J. C. A. Romanovsky, and P. Alencar, editors, *Advances in Software Engineering for Multi-Agent Systems*, pages 220–237. Springer-Verlag, 2004.

[7] H. Kitano and S. Tadokoro. Robocup rescue: A grand challenge for multiagent and intelligent systems. *AI Magazine*, 22(1):39–52, 2001.

[8] J. Niu, K. Cai, S. Parsons, P. McBurney, and E. Gerding. What the 2007 TAC market design game tells us about effective auction mechanisms. *Journal of Autonomous Agents and Multi-Agent Systems*, 2010.

[9] I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: a tool for research on multi-agent systems. *Applied Artificial Intelligence*, 12:233–250, 1997.

[10] M. J. North, N. T. Collier, and J. R. Vos. Experiences creating three implementations of the repast agent modeling toolkit. *ACM Trans. Model. Comput. Simul.*, 16(1):1–25, 2006.

[11] M. Oey, S. van Splunter, E. Ogston, M. Warnier, and F. Brazier. A framework for developing agent-based distributed applications. In *WI-IAT '10: Proc. IEEE/WIC/ACM Int. Joint Conference on Web Intelligence and Intelligent Agent Technology*, pages 470 –474, 2010.

[12] E. Ogston and S. Jarvis. Peer sampling with improved accuracy. *Peer-to-peer Networking and Applications*, 2(1):51–71, 2009.

[13] R. Vincent, B. Horling, and V. R. Lesser. An agent infrastructure to build and evaluate multi-agent systems: The java agent framework and multi-agent system simulator. In *Revised Papers from the International Workshop on Infrastructure for Multi-Agent Systems*, pages 102–127. Springer-Verlag, 2001.

[14] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, 2002.

[15] N. J. E. Wijngaards, B. J. Overeinder, M. van Steen, and F. M. T. Brazier. Supporting internet-scale multi-agent systems. *Data and Knowledge Engineering*, 41(2-3):229–245, 2002.

[16] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3:285–312, 2000.