Mobile agent approach to networked robots

2 Background

2.1 The ALLIANCE architecture

Figure 1 shows a simplified diagram of ALLIANCE. Its main components are motivational behaviours (MBs) and behaviour sets (BSs). MBs receive input from inter-robot communication and sensors, and are linked to an associated BS. A BS receives input from sensors and provides output to the robots’ actuators. It contains code that controls the behaviour of a robot at runtime. Each BS relates to an independent activity.

Each MB determines whether its associated BS is selected at runtime by calculating the robot’s motivation to perform it. Motivation in ALLIANCE is calculated using the mathematical formula shown in Fig. 2. A number of parameters affect the value of motivation at time t. These parameters are:

- **Impatience** determines the willingness of a robot to conduct a BS. This is affected by the activity of other robots, in that, if some other robot is conducting this behaviour at this time, the robot will be less impatient to conduct the BS.
- **Activity suppression** ensures that a robot selects only one BS at a time to perform by suppressing the motivation to perform other BSs once one BS has been selected.
- **Sensory feedback** used to establish if a particular behaviour is currently applicable, based...
When a robot decides to acquiesce a BS, it sets the value of acquiescence to 0, causing the motivation for the BS to return to 0, while deactivating activity suppression. This allows another MB to increase its motivation beyond the threshold of activation and select a new behaviour. Whenever a BS is selected, a robot broadcasts this information to all other robots at a set rate to inform them of its activity.

2.2 Networked robots

Networked robots use distributed computing hardware and software to allow communication between disparate components. Such systems usually employ Internet protocols such as TCP/IP or UDP/IP for communication and wireless LAN hardware. Networked robot systems exist which represent the full spectrum of robot systems from single tele-operated robots, such as the University of Essex Tele-robot [3], to multiple autonomous robot systems, like MURDOCH [4]. The unifying characteristic of these systems is their use of networking technology for communication.

2.3 Distributed computing paradigms

A number of distributed computing paradigms exist with which distributed computing systems, such as multiple-robot systems, can be structured. These include:

Client/server In this paradigm (Fig. 3a), a server has the knowledge and resources to perform a task. A client sends a message to the server requesting task execution. The server performs the task using its knowledge and resources, and returns a result to the client.

Remote computation In this paradigm (Fig. 3b), a client has the knowledge, while a server has the resources to perform a task. For the client to obtain the result to a task, it must send its knowledge to the server. The server uses this knowledge and its own resources to perform the task before returning a result to the client.

Code on demand In this paradigm (Fig. 3c), a client has the resources, while a server contains the knowledge to perform a task. For the client to obtain a result to a task, it
must send a message to the server to request knowledge. The client uses the server’s knowledge and its own resources to perform the task and obtain a result.

**Mobile agents** In this paradigm (Fig. 3d), an agent server is the host to a mobile agent which contains the knowledge and results from previous tasks, but not the resources it requires to perform a new task. Another agent server contains the required resources. Instead of forwarding knowledge to the new agent server, the mobile agent moves to access the resource. When the mobile agent has completed its task, it can remain at the new server or move to another agent server carrying its knowledge and task results.

### 3 Mobile agents

In the previous section, we compared activity in a variety of distributed computing paradigms. In this section, we examine mobile agents in more detail.

#### 3.1 Execution environment

Mobile agents exist within an execution environment, provided by multiple agent servers, as shown in Fig. 4. Agent servers provide areas known as places, in which an agent can reside and interface with functionality provided by the server and host computer. Most agent servers and mobile agents are written in Java, which is an interpreted computer language, and can therefore execute on a variety of heterogeneous computer platforms.

Multiple host computers providing a distributed (but unified) mobile agent execution environment each host a single agent server, which can be uniquely identified using the host’s IP address. Agent servers may, however, contain multiple places, between which, multiple mobile agents can simultaneously migrate. Basic mobile agents are lightweight computing components, because many of the functions which they execute are provided by agent servers in a function library, e.g. functions for agent creation, destruction, cloning, migration and fault-tolerant storage to persistent media, such as a computer hard drive. Function libraries allow developers to concentrate specifically on the wider purpose of the mobile agents in their system rather than on their basic execution requirements.

Most agent servers make extensive use of multi-threading in order to execute agent activity. Mobile agents usually run in an independent thread, allowing multiple agents to execute concurrently. Mobile agent developers can also make use of multithreading within individual agents to allow simultaneous activity to be executed, e.g. agent communication, while also conducting high-level planning etc. Mobile agents can make use of many communication mechanisms, including sockets, RMI and CORBA. Although remote communication using these mechanisms is possible, migration and local communication are more often employed in mobile agent systems because this helps to reduce network load in comparison to other distributed computing paradigms, which must all communicate remotely across networks.

#### 3.2 Characteristics and functionality

Mobile agents are goal-driven and autonomous; they can be reactive, adaptable, dynamic, temporally continuous, operate asynchronously, communicate and learn. Because they can provide all of these characteristics and also move from place to place, mobile agents can provide the functionality found in all other distributed computing architectures combined. In addition, they provide a more natural design philosophy for distributed computing systems when compared to other distributed computing paradigms, i.e. traditional client/server-based architectures require more complex interconnected networks of heterogeneous computing nodes than mobile agents. Using mobile agents, we do not need to take into account network connections and consider all distributed computing nodes as offering equivalent functionality. This allows the development of systems which are more likely to be logically sound and robust.

Mobile agents can be used to produce intelligent, dynamic, fault-tolerant and scalable distributed systems and can reduce network load. This has allowed them to be applied in a range of application areas [5], including computational outsourcing, dynamic network management, load balancing, personalised user presence, software deployment, intelligent data, temporary applications, application of dynamic protocols and intelligent remote action [6].

### 4 Implementation

Having provided some background information about ALLIANCE, networked robots and mobile agents as a distributed computing paradigm, this section describes our implementation of ALLIANCE in a mobile agent envi-
In order to conduct rapid development and perform preliminary experiments, we initially developed our ALLIANC3e implementation using simulated robots. This section begins with its examination, before a description is subsequently provided of our real robot implementation.

4.1 Simulated robot implementation—hardware and software

In our simulated robot system, we implemented ALLIANC3e with the mobile agent hardware and software shown in Fig. 5. This implementation was conducted because simulated robots allow rapid development and testing relative to real robots, as they are not prone to hardware failures nor require battery recharging. Without the use of real robots, it is, however, not possible to fully and accurately test an implementation in a real-world environment, hence, the subsequent development of the real robot system.

The simulated robot system contained four networked PCs, each running Windows XP operating system and containing the Java 1.4 programming language. The Grasshopper 2 (GH2) agent platform [7] was used to host mobile agents, the timing server to time experiments and an Apache Web Server v2.0 to serve Java mobile agent .class files to the GH2. ActivMedia’s Pioneer simulator [8] was used to simulate Pioneer 2DX robot functionality on three of the PCs to represent a networked multiple robot team.

GH2 is a free OMG MASIF-compliant Java-based mobile agent server which provides an execution environment and function library for Java-based mobile agents. The function library provides functions for the creation, destruction, cloning, migration and persistent storage of agents. A mobile agent developer overrides a live() function, in which they incorporate their mobile agent’s runtime behaviour. The Apache Web Server was used to store the agent .class files. The GH2 agent servers obtain .class files from this central repository.

ActivMedia’s ARIA robot control software was used to control the low-level behaviour of robots. It allows various levels of control to be executed on the ActivMedia Pioneer 2DX robots (simulated or real). It is written in C++, but is provided with a Java wrapper, which allowed commands to be written directly in our Java-based mobile agents. The timing servers were multi-threaded Java-based servers, which listened on known ports and registered the time of any connection. Because it is not possible to precisely synchronise the internal clock of distributed computers, we employed a single timing server during each experiment to generate consistent times for experiment duration calculation.

4.2 Real robot implementation—hardware and software

In our real robot system, we implemented ALLIANC3e with the mobile agent hardware and software shown in Fig. 6. This environment contained two PCs, one running Windows XP and the other running Linux. The PC running Windows XP contained Java 1.4, GH2 (to host mobile agents), a timing server (to time experiments) and the Apache Web Server v2.0 (to serve Java mobile agent .class files to the GH2). The PC running Linux contained SSH to allow remote login and SFTP to allow the downloading
of files to three ActivMedia Pioneer 2DX mobile robots (each running Linux and were network-accessible via wireless LAN). Each robot contained Java 1.3, GH2 and ActivMedia’s ARIA robot control software.

4.3 Architecture

Our mobile agent implementation of ALLIANCE was initially developed and tested in the simulated robot system; Fig. 7 shows its main components. An ALLIANCE control agent (ACA) is a mobile agent which engages in three main activities: calculation of MBs, execution of BSs and Inter-robot Communication (IC).

To conduct IC, calculate MBs and execute BSs concurrently, we implemented these activities as subclasses of the thread class. We started each thread in the ACA’s live() function, allowing them to execute simultaneously. Using synchronised methods, these classes are able to access shared variables within the main body of the ACA, allowing coordinated behaviour. As these activities operate independently, the main agent thread is free to interact with the server and other agents. The MB thread calculates a motivation value for each BS every 100 ms. The BS contains code for the execution of any BS selected by the MBs in the MB thread and executes every 100 ms.

For IC, the IC thread creates and interacts with a second form of mobile agent, an ALLIANCE communication agent (AComm). At each cycle of the IC thread, i.e., every 4 s, the IC thread on each robot creates a single AComm. This agent is created with a list of all known robot addresses, the ID of the creating robot and the behaviour of the creating robot. Upon creation, this agent automatically creates a copy of itself on all the other robots in its list and then removes itself from the creating robot.

After creating the AComm, the IC thread uses the agent server to locate and interact locally with any AComms which have been copied to this robot from other robots. It obtains the ID information and current behaviour of any AComms currently residing on the robot, thereby, obtaining information on the activity of all other communicating team mates. After providing the local IC thread with their information payload, the AComms remove themselves from the robot host in order to prevent a build-up of agent numbers and resource use. The IC, conducted in this way, allows the MB thread to be provided with relevant...
information with which to calculate motivation values for behaviour selection.

5 Experiments

Using the simulated robot system shown in Fig. 5, the real robot system shown in Fig. 6 and the mobile agent implementation of ALLIANCE shown in Fig. 7, two sets of experiments, adaptability and fault-tolerance, were conducted. A description of the activities involved in the real robot experiments is provided in the following sections. The simulated experiments involved a similar set of activities, but were less automated in their static-code experiment setup, as highlighted in Sect. 6.

5.1 Adaptability experiments

Adaptability experiments were designed to investigate how mobile agents might affect the adaptability characteristics (i.e. the ability to update or adapt system code at runtime) of an ALLIANCE-controlled multiple-robot system. One setup used an ALLIANCE mobile agent implementation (in which mobile agents were employed to allow existing ACAs to be dynamically updated by new ACAs at runtime), whilst the second investigated the updating of a traditional static-code ALLIANCE implementation (using remote robot login, SSH, and file transfer using SFTP).

Figure 8a shows mobile agent adaptability experiment activities. Initially, existing ACAs are used to control multiple robots via robot agents (RAs) (1). A GH2 platform is then started, which automatically creates a new ACA (2). The new ACA clones itself to all robots, at which, the clones communicate with existing ACAs, causing them to remove themselves from the robot (3). The new ACAs then connect to the robot via RAs before communicating with a timing server (4).

The timing of an experiment trial begins on the creation of the GH2 platform in step 2 and ends when the last new ACA communicates with the timing server in step 4.

Figure 8b shows static-code adaptability experiment activities. Initially, existing alliance control programs (ACPs) are used to control multiple robots (1). SFTP is then used to download new ACPs (3) and SSH used to stop existing ACPs (2). SSH is then used to start the new ACPs, which connect to their robots and communicate with a timing server (4). The timing of an experiment trial begins on the creation of the first SFTP connection to robots in step 2 and ends when the last new ACP communicates with the timing server in step 4.

5.2 Fault-tolerance experiments

Fault-tolerance experiments were designed to investigate how mobile agents might affect the fault-tolerance characteristics (i.e. the ability of a system to remain operational in the presence of faults) of an ALLIANCE-controlled multiple-robot system. One setup used an ALLIANCE mobile agent implementation (in which an existing ACA is able to autonomously migrate from a failing robot to a secondary robot in order to maintain mission level data, before migrating to a new replacement robot), whilst the second investigated a traditional static-code ALLIANCE implementation (in which a failing robot loses mission-level data when it fails and is replaced using a new robot with no previous mission experience).

Figure 9a shows mobile agent fault-tolerance experiment activities. Initially, an ACA executing on Robot 1 detects a terminal fault and communicates with a timing server (1). This ACA then migrates to Robot 2 (2) and waits until a robot start-up agent (RSA) communicates with it to show that a new Robot 1 has been started (3). The ACA waiting on Robot 2 then migrates to the new Robot 1, connects to the robot via an RA and communicates with the
timing server once more (4). The timing of an experiment trial begins on the connection of the ACA to the timing server in step (1) and ends on the ACA connection to the timing server in step (4).

Figure 9b shows the static-code fault-tolerance experiment activities. Initially, an ACP executing on Robot 1 detects a terminal fault and communicates with a timing server (1). SFTP is then used to download a new ACP to a new Robot 1 (2). SSH is then used to start this new ACP (3). The new ACP connects to the new Robot 1 and then communicates with the timing server (4). The timing of an experiment trial begins on the connection of the ACP to the timing server in step (1) and ends on the ACP connection to the timing server in step (4).

### 6 Experimental results and analysis

This section examines the results of the experiments conducted using the simulated and real robot systems described in Sects 4 and 5.

#### 6.1 Adaptability experiments

Adaptability experiment results are shown in Fig. 10a,b and Table 1a,b. The adaptability experiments show that it is considerably quicker and easier to update and adapt the control software on multiple robots using mobile agents in comparison to a traditional method using static code.

![Graphs](image-url)

**Fig. 10** Adaptability experiment results charts. a Simulation results. b Real robot results.
The divergence between the simulated and real robot results is due to the use of scripts in the real-robot static-code experiments, which were used to reduce or remove human performance factors and speed up and automate the updating process. Despite this, it can be seen that mobile agents in our experiments update code much more quickly than static code in either system.

Mobile agent performance is an improvement over traditional methods because agent creation occurs in parallel on all robots, while in the static-code implementation, activities occur in series.

Due to this parallel execution, as robot team numbers increase, the mobile agent to static-code time ratio should, theoretically, widen, i.e. mobile agents relative to static code should be ever more effective as the robot number increases.

### Table 1 Adaptability experiment results tables

| Type\|trial | 1  | 2  | 3  | 4  | 5  |
|------|-------|----|----|----|----|----|
| MA 1 | 6.2   | 6.1| 6.1| 6.1| 6.0|
| SC 1 | 21.7  | 26.5| 22.2| 26.1| 21.4|
| MA 2 | 6.3   | 6.1| 6.4| 6.1| 6.2|
| SC 2 | 59.1  | 55.6| 73.2| 60.5| 61.3|
| MA 3 | 6.4   | 6.3| 6.3| 6.0| 6.2|
| SC 3 | 96.5  | 84.5| 89.2| 93.2| 79.2|
| MA 1 | 8.1   | 6.6| 6.6| 6.9| 7.8|
| SC 1 | 26.0  | 23.7| 26.2| 25.8| 24.8|
| MA 2 | 8.5   | 9.2| 8.0| 8.7| 7.8|
| SC 2 | 36.8  | 38.6| 35.4| 38.8| 38.2|
| MA 3 | 10.5  | 10.4| 10.8| 8.9| 10.7|
| SC 3 | 52.7  | 52.7| 50.2| 52.6| 54.8|

### 6.2 Fault-tolerance experiments

The fault-tolerance experiment results are shown in Fig. 11a, b and Table 2a,b. The fault-tolerance experiments have shown that, by using mobile agents, it is possible to retain the state and data of a failing robot.

As in the first set of experiments, scripts were used to automate the static-code experiments where possible. This increased the speed with which replacement robots were started in the real robot experiments, meaning that more time (approximately 7–8 s) was required when using mobile agents to start up a new replacement robot compared to the static-code implementation. Although this process took longer using mobile agents in the real robots, the traditional static-code implementation in both simulated and real robot experiments lost robot state and data when...
the robot failed, while such data was retained in the mobile agent implementation.

7 Conclusions and future work

In this paper, we have examined an implementation of the ALLIANCE architecture in simulated and real networked robots using mobile agents, in which we extend adaptability and fault-tolerance. These functional extensions arise as a result of the implementation of ALLIANCE in the mobile agent environment, and not through its architectural adaptation. These extensions could equally be achieved with other distributed mobile robot control architectures.

Mobile agents offer many opportunities for the extension of functionality in multiple-robot systems, as they unify the functionality found in all other distributed systems and provide a natural design philosophy for distributed systems, such as multiple-robot architectures (based upon an equal allocation of basic functionality throughout all mobile-agent-server-enabled system components). In conjunction, these characteristics can allow multiple-robot system developers to employ mobile agents as a tool to develop robust cohesive architectures in which system structure can be easily extended by making use of in-built functionality available in the mobile agent environment. We are using this implementation of ALLIANCE as the control selection mechanism of a multiple tele/autonomous robot architecture [9], in which we aim to make further use of the beneficial functionality which mobile agents offer.

References

Conditions for through-process analysis of product surface in steel manufacture

Received: 1 September 2004 / Accepted: 3 November 2004 / Published online: 17 November 2005
© Springer-Verlag London Limited 2005

Abstract In the steel industry, there is a trend to equip an increasing number of production lines with camera-based inspection systems to monitor the surface quality of the steel. These systems are dedicated to automatically detect and classify surface defects. When camera systems are used in different processing lines through which the steel subsequently passes, it becomes possible to track the surface quality through the chain. Linking through-process data enables fast root cause analysis and optimisation of manufacturing practice to minimise the defect level. The evaluation of the changes in surface condition through the process is hampered by problems in data synchronisation, changes in product geometry (sealing), parts of the product that are taken out of the process chain (slit edges, cut off heads or tails) and delays in the processing chain (e.g. due to the maintenance of a production line). This paper highlights the conditions and problems encountered in integrated process analysis in an industrial environment.

1 Introduction

Two important markets for Corus in The Netherlands are the automotive industry and the packaging sector. Car manufacturers use steel for the structural frame, chassis (the “car body”), engine, wheels, etc. A common product delivered to the automotive industry is galvanised strip steel, i.e. flat material of 1.80–2.20-m wide, 0.6–1.6-mm thick and 1–2-km long, that is wound onto coils of, typically, 20 tonnes. In packaging, steel is used to manufacture cans for food and beverages, as well as aerosol spray containers. A common product type supplied to this market are coils of tin-coated steel, ~1-m wide, ~0.2-mm thick and, typically, 6-km long.

Galvanised strip steel is used to make external body parts, such as roofs, doors and bonnets. Obviously, these external parts require steel surfaces that are absolutely free from surface defects like scratches, roll imprints, pits and oxide striper. Similarly, the packaging industry requires their products to be absolutely defect-free. Critical material produced at the steel manufacturer is, therefore, inspected for its surface quality.

In the past, the task of visual inspection was mostly carried out via manual inspection by the operators or inspectors. Evidently, the change in the manning of a manufacturing line, the changing degree of attentiveness of the operator and variations in the perception of the relative importance of errors introduce a substantial level of subjectivity into this procedure. The advent of fast cameras, high-bandwidth data transport and increased computational power has provided the possibility to fulfil automated inspection of the surface by computerised camera-based vision systems. These vision systems, or surface inspection systems (SISs), perform image capture, image processing and automated defect detection and classification. Operators usually have the possibility to view the statistics of defect occurrence over a given production period and they can retrieve the images of individual defects, which are stored as regions of interest (ROI).

Generally speaking, SISs are installed at the end of a production line, but in certain cases, these systems inspect the incoming product. Since the manufacturing of galvanised steel is performed in subsequent production phases, like hot rolling, pickling, cold rolling, annealing and galvanising, and each line can be equipped with one or more SISs, this creates the possibility to study the evolution of the surface quality through the production chain.