Blackboard Architecture for Intelligent Embedded Systems

Kum Wah CHOY

A thesis submitted in partial fulfilment of the requirements of Nottingham Trent University for the degree of Doctor of Philosophy

February 2005

Abstract

It can be argued that the future of machines lies in embedding intelligent systems within them. Unfortunately, sophisticated artificial intelligence (AI) software is usually large and complicated, requiring powerful processors that are not practical in most embedded systems owing to cost, size, and heat production. One solution is to distribute the intelligent processing across many less powerful processors. This research has investigated the suitability, characteristics, and potential of a distributed blackboard system as an architecture for the implementation of complex AI software in an embedded distributed processing network. A distributed blackboard system called DARBS (Distributed Algorithmic and Rule-based Blackboard System) has been implemented on a distributed processing network comprising up to 18 personal computers. Using the TileWorld environment as a test-bed, the distributed set-up was found to outperform a non-distributed one, although both were non-optimal. The speedup factor of the distributed blackboard system increases up to a maximum as the number of agent processors (APs) increases, after which it drops a little and levels off. To obtain maximum speedup for a given number of agents, it was found that an even distribution of agents across the APs is required, while avoiding saturation of the blackboard. The optimum number of agents per AP was found to be two, with blackboard saturation starting at eight APs. Based on these findings, an embedded version of DARBS (called emDARBS) was designed and implemented on a SARNet parallel processing network of low-cost StrongARM processors. This implementation has demonstrated that a distributed blackboard system is suitable for embedded distributed processing networks, but that some changes are required to tailor it for embedded systems. The potential for embedding intelligent systems in everyday machines has thus been demonstrated.

Acknowledgements

I would like to take this opportunity to thank my supervisors, Prof. Adrian Hopgood and Dr Lars Nolle, and my advisor, Prof. Brian O'Neill for their guidance and support throughout my PhD research. Thank you also for taking the time to read and giving me feedback on my thesis.

I would also like to thank my dad, mom, and sister for their motivational discussions and emotional support. Without them, I would not have lasted as long as I did. To my girlfriend, Joyce, thank you for being patient and understanding during my bad times. Thank you for being there when I needed you.

To my friends and colleagues, thank you for being there to bounce my ideas off. Our informal discussions really helped me a lot. Also to Sharron, thank you for your immeasurable help with my English. I really appreciate it.

Finally, to those that I have unintentionally missed out, thank you.

Table of contents

List of figures

1. Introduction

People these days are demanding more intelligence from their machines, from small handheld devices to cars and airplanes. They want machines that are intelligent enough to work out what they want from them. They also want machines that are able to constantly adapt to their ever-changing requirements and needs. Nowadays, these machines contain at least one processor or micro-controller embedded within them. These machines are said to have embedded systems in them. The intelligence of future machines lies in their embedded systems. In order for machines to be intelligent, their embedded systems need the intelligence to know what to do and when to do it, based on their sensory data and knowledge of the user. These types of intelligent embedded systems would more easily fuse into society than embedded systems that require the user to have a deep technical knowledge. Embedded systems should be able to learn and adapt to the user instead of the user learning and adapting to it [1]. Therefore, the future of machines, such as mobile phones, washing machines, watches, televisions, industrial robots, cars etc. lies with its intelligent embedded systems.

1.1 A distributed blackboard system approach

To achieve the future vision of intelligent embedded systems, complex intelligent software is required. This type of intelligent software is usually large, complicated and processor-intensive, thus requiring powerful processors. Unfortunately, in embedded systems, processing power is a constraint. As embedded systems become more intelligent and require less human intervention, the reliability and robustness of the systems also become constraints, especially for life-critical systems and/or mission

critical systems. Example areas of these embedded systems include: space exploration, military, aviation and health care. Embedded systems have many other types and degrees of constraints and not all constraints can be satisfied at the same time. A detailed discussion about embedded systems is in section 2.1.

One method of increasing the complexity and intelligence of embedded systems while still meeting the requirements and constraints of the embedded systems is to distribute the intelligent processing across many less powerful processors. These processors can be networked together so that they can share their information as they work towards their common goal. Distributing the processing would increase the overall processing power and at the same time make it possible to have redundancy, thus increasing the reliability and robustness of the overall system. Usually, intelligent machines are made up of modules of intelligent embedded systems and therefore would fit very well in a distributed processing network. A typical structure for a distributed processing network can be seen in Figure 1.

It could be argued that increasing the number of processors used could increase the chances of hardware failure. However, it is argued that the overall system can be better tested in small modules compared to a single, large and complex system, thus reducing the chances of failure. A detailed review of distributed processing networks is given in section 2.4.

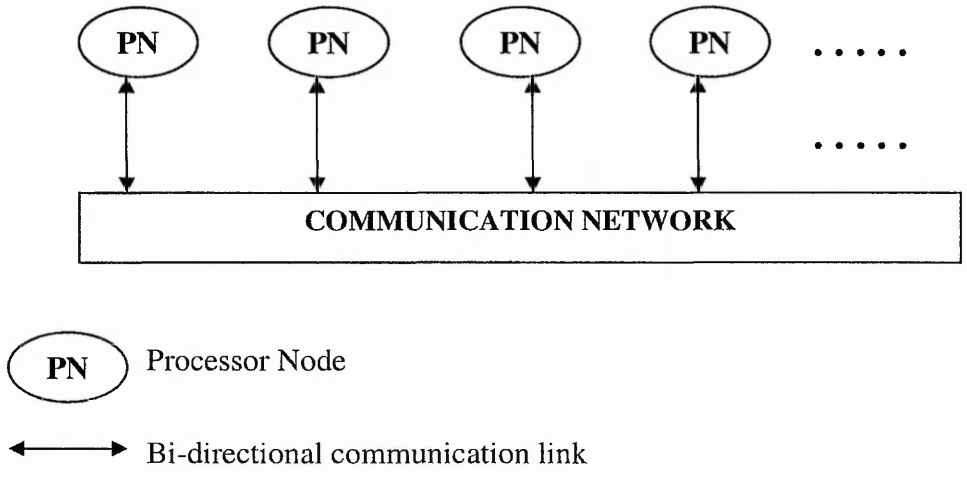


Figure 1. Typical structure of a distributed processing network

On the software side, in order to exhibit complex intelligent behaviour, a variety of artificial intelligence (AI) methods (e.g. neural network, rule-based system, genetic algorithm, etc.) working together as one system is required. This is because no one AI method is suitable for solving all AI problems. For example, neural networks are particularly suitable for solving pattern classification problems especially if the patterns to be classified are not known to the designer [2]. Rule-based systems, on the other hand, are suitable to solve problems where the domain knowledge of the problem is fully known and can be easily coded into rules on to the system [3]. Genetic algorithms are suitable for optimisation problems, especially if the search space is of multiple dimensions [4]. Therefore, when complex intelligent behaviour is needed to solve a problem, an artificial intelligence architecture that can synergise all these different AI methods together in a distributed processing network is required. One such architecture that fits perfectly to this distributed processing network is the blackboard architecture [5]. The blackboard architecture, also known as blackboard system, is analogous to a team of experts who communicate their ideas by writing them on a blackboard. The team of

experts, in this case, consists of the different AI methods working together to solve the overall problem on the blackboard. Figure 2 shows how a blackboard system would fit on a distributed processing network. A detailed review of the blackboard architecture is given in sections 2.2.1.

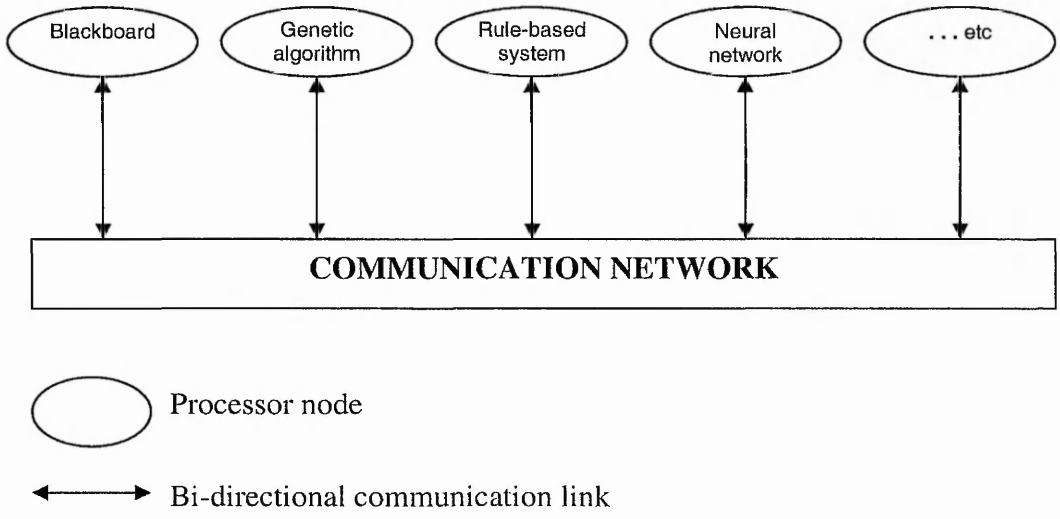


Figure 2. Distributed blackboard system on a distributed processing network

There are limited studies of using the blackboard system approach for distributed embedded processing networks and therefore there is a need to investigate the suitability and potential of this approach for distributed embedded processing networks and embedded systems in general. The challenges that arise from implementing a blackboard system in a distributed embedded processing network will also need to be investigated.

1.2 Aims of this research

This research aims to investigate the suitability, potential and characteristics of a blackboard system in a distributed embedded processing network. This research would provide the foundation for future implementation of intelligent embedded systems. So

far, traditional blackboard systems have all been implemented in a single processor system with each expert taking turns to be executed. This is because in a single processor system only one instruction can be executed at a time. A controller determines which expert will be executed next, based on the information that is on the blackboard. More recently, distributed blackboard systems have been developed that make use of the inherent parallelism of the blackboard system. The University of Edinburgh previously had a research project to use a distributed blackboard system in a distributed processing network [6]. A Canny edge detector algorithm [7] was implemented in a distributed blackboard system running on a Meiko multi-transputer system. In the research, it was discovered that the distributed blackboard system approach created larger overheads which in turn reduced the parallel performance of the algorithm. However, blackboard systems provide greater implementation flexibility in terms of being able to implement different modules known as knowledge sources (see section 2.2.1). In this thesis, even though maximum parallel performance cannot be achieved, blackboard systems are selected because of their implementation flexibility.

The performance of a distributed blackboard system has been investigated in a NCUBE/10 hypercube architecture [8]. In that research, the blackboard system is implemented in fine-grain parallelism [9], where the blackboard and the knowledge sources are split into many parts to run on separate nodes. This type of fine grain parallelism is good for maximising speedup but complicates the implementation of the blackboard system. In this thesis, coarse-grained parallelism [10] is chosen instead as it will allow the programmer to concentrate more on knowledge implementation and ignore issues of fine-grained parallelism. It is suggested that the increase in performance of the blackboard system in a distributed processing network would be 5 to 10-fold

compared to a blackboard system in a single processor [11]. This increase in performance would need to be tested and analysed to see if it is suitable for an intelligent embedded system.

Implementing distributed blackboard systems for embedded systems has been proposed before, for example, a distributed blackboard architecture to support multi-agent systems [12]. In that paper, it is suggested that multi-agent systems would be needed to implement future intelligent systems, and the distributed blackboard architecture is the ideal architecture to implement this. This thesis investigates the benefits of multiple agents in a distributed blackboard system.

The research aims can be summarised as follows:

- To investigate the suitability, potential and characteristics of a distributed blackboard system in a distributed embedded processing network by:
    - First, implementing a distributed blackboard system in a distributed processing network (chapter 3).
    - Then, running performance experiments and evaluating the suitability, potential and characteristics of the distributed blackboard system in the distributed processing network (chapter 4).
    - Finally, based on the results of the performance experiments, implement the distributed blackboard system on a distributed embedded processing network. The actual distributed processing hardware chosen is not important as long as the behaviour of the distributed blackboard system can be investigated on a truly parallel platform (chapter 5).

- To investigate the challenges that arise from implementing a distributed blackboard system in a distributed embedded processing network (chapter 5 & 6).
- To evaluate critically the effectiveness and feasibility of a distributed blackboard implementation in a practical embedded application (chapter 6).

1.3 Summary

The future of machines in general lies in their intelligence. In the future, intelligent machines that are made up of many intelligent embedded systems would be able to constantly adapt to the ever-changing user's requirements. These intelligent embedded systems require complex intelligent software in order to exhibit this type of intelligent behaviour. These types of complex intelligent software are usually processor intensive. Unfortunately, embedded systems usually have, amongst others, processing power constraints and as such distributing the intelligent software across many less powerful processors is one way of reducing the processing power constraint. Complex intelligent software is usually made up of many different types of artificial intelligence techniques (e.g. neural network, rule-based system, genetic algorithm, etc.) all working together to solve a common problem. The blackboard architecture is a suitable architecture for different artificial intelligence techniques to work together in a distributed processing network.

The aim of this research is to study the suitability, potential and characteristics of a distributed blackboard system in a distributed embedded processing network. The performance of the distributed blackboard system in the distributed processing network will be investigated. The challenges that arise from implementing the distributed blackboard system in a distributed embedded processing network and how this would affect future implementation of embedded distributed blackboard system will be studied.

2. Literature survey

This chapter gives a detailed overview of the subject areas that this research covers. The areas that are discussed are: embedded systems, artificial intelligence systems, intelligent embedded systems, distributed processing networks and distributed blackboard systems in distributed processing networks.

2.1 Overview of embedded systems

Nowadays, the miniaturisation of electronic components has made it possible for an entire computer system to be fitted onto a single integrated circuit (IC). This has led to the development of embedded systems. Embedded systems are generally small self-contained specific-purpose systems that comprise of both hardware and software [13]. They are included in, for example, mobile phones, watches, microwave ovens and photocopiers. A general block diagram of an embedded system is shown in Figure 3. Embedded systems take in data from their environment either in the form of sensory data or communication data from other embedded systems. They then process the input data according to their software and produce output in the form of display or data to be further transmitted to other embedded systems. Some embedded systems use direct memory access (DMA) [14] for collecting and sending input and output. These days, complex embedded systems are made up of a few embedded systems connected together. For example, a car's engine management system, anti-lock braking system, temperature control system and car navigation system are all connected to the driver's display system.

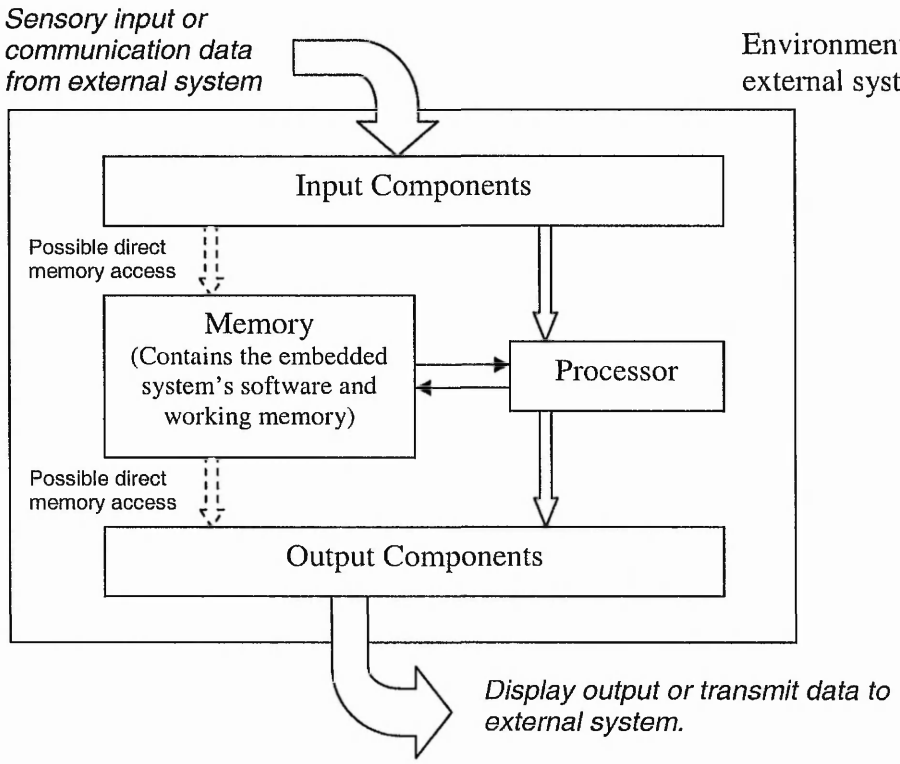


Figure 3. General block diagram of an embedded system

Embedded systems differ from normal general-purpose computers in that they generally have tighter constraints. The main constraints of embedded systems are cost, processing power, memory capacity, reliability, electrical power consumption, physical size, real-time and working environment. These constraints will be explained in the following sections.

Implementations of embedded systems vary greatly as each embedded system has different degrees of constraint. As such, there is no single architecture that all embedded systems can use. For example, the cooling control system for a nuclear power plant has tight real-time and reliability constraints [15] whereas an electronic dictionary has tight cost and physical size constraints. The nuclear power plant would use a distributed

processor architecture to meet the real-time and reliability constraints [16] whereas the electronic dictionary would use a low processing power processor to keep the cost down.

For the envisioned future embedded systems that are able to learn and adapt to the user's needs, complex artificial intelligence software is required. However, complex artificial intelligence software is usually processor intensive and this is in conflict with the processing power constraint. Therefore, there is a need for artificial intelligence software that is not processor intensive yet complex enough to be able to learn and adapt to the user's needs.

2.1.1 Cost

The main constraint of any commercial embedded systems is cost. This includes both development and manufacturing costs. One way of reducing the total cost of an embedded system is to use the cheapest possible components that meet the minimum specifications. Another way of reducing development costs is to use off-the-shelf modules (both hardware modules and software modules). This is because off-the-shelf modules have been tried and tested by other manufacturers and are less likely to incur extra testing and verification costs. An example of low cost component design is in the PicoRadio networks [17]. PicoRadio network components provide low-cost, low-powered wireless communication for embedded systems.

2.1.2 Processing power

The processing power constraint is mainly governed by the price of the processor and the amount of electrical power consumed. Processing power is commonly measured in

MIPS (millions of instruction per second) [18] and greater MIPS values mean more powerful processors. Usually, the more powerful the processor, the more electrical power (in terms of Watt) it consumes. Another consideration for processing power is the number of bits the processor processes simultaneously (e.g. 8-bit, 16-bit, 32-bit, or 64-bit processor). In general, 64-bit processors with high MIPS are more expensive than 8-bit processors with low MIPS. For example, the MIPS for a 32-bit 200MHz StrongARM SA-110 RISC processor according to the Dhrystone benchmark [19] version 2.1 MIPS is 230 [20].

2.1.3 Memory capacity

The memory capacity available in an embedded system is also governed by the cost of the memory. Although the price per memory capacity (measured in terms of cost/megabyte) has decreased significantly throughout the years, in general, bigger memory capacity costs more than smaller memory capacity. As such, embedded systems are designed to use the minimum possible memory capacity. This usually means that the embedded system software has to be as compact as possible. There is on going research to develop compact embedded system software and an example of this is PicOS, the operating system for extremely small embedded platforms [21].

2.1.4 Reliability

For embedded systems that are used in life-critical and/or mission-critical systems, the reliability of these systems becomes an important constraint. These types of systems cannot afford to fail as failure could cost lives. Reliability of embedded systems is a widely researched topic because of its importance and it can be broadly broken down

into two parts: the reliability of the hardware and the software. Formal design methodologies for both hardware and software are being developed to increase the reliability of the developed embedded system [22]. Another way of increasing the reliability is to introduce redundant systems. This means that if one system fails, the next redundant system will takeover. Usually the other redundant systems are designed by different teams and use different hardware to reduce the chances of producing the same error [23]. An example of this is in the Boeing 777 aircraft [23]. This aircraft uses three separate embedded systems for its Primary Flight Computer with two of them acting as redundant embedded systems. These two redundant systems act as backup for the first system.

2.1.5 Electrical power consumption

Lately, embedded systems are extensively used in portable devices. These portable devices run on limited battery power. In order to prolong the mean time between battery charges, the electrical power consumption of these types of embedded systems is kept to a minimum. One way to keep electrical power consumption low is to use slower processors or processors with dynamic power management (DPM) capabilities [24]. There is on-going research in power management for embedded systems, both on the hardware side [25] and on the software side [26]. More recently, work has been done to model power mode dependency and mode selection for power-aware embedded systems [27]. This has led to a more efficient use of available power on an embedded system. In general however, to keep the processor's electrical power consumption low, the processor's clock is slowed down. This in turn leads to low processing power (i.e. low MIPS). Therefore, a balance needs to be attained between processing power and electrical power consumption. As an example of typical power consumption, the 32-bit

200MHz StrongARM RISC processor consumes 330mW when operating in normal mode [20].

2.1.6 Physical size

As embedded systems are extensively used in portable devices, the physical size of an embedded system also needs to be small. This is so that the overall size of the device is small and portable. The physical size constraint will make the designer choose components that are small and energy-efficient, i.e. components that do not produce much heat. Components that produce too much heat will need large heat sinks which will cause the overall size of the device to be large. One of the main components that generate a lot of heat is the processor. In general, fast powerful processors generate more heat as they consume more electrical power than slower less-powerful processors. The physical size constraint also limits the number of components that can be fitted on an embedded system and because of this, new technologies such as System-On-a-Chip (SoC), have been developed to embed all the components into a single integrated circuit [28].

2.1.7 Real-time

Most embedded systems used to control processes in the real world have real-time constraints. Real-time embedded systems are embedded systems that can respond to the changes in the environment faster than the environment can change. Real-time embedded systems have time deadlines which usually coincide with the frequency of the changes in the environment. These types of embedded systems also tend to use very fast processors or many slower processors connected together, with each processor

controlling a small section of the overall process. Examples of real-time embedded systems are flight control systems [29], nuclear power plant control systems [30] and missile targeting systems [31].

2.1.8 Working environment

Some embedded systems are used in hazardous environments such as those in space exploration and deep sea exploration. These types of embedded systems have a working environment constraint. For example, the Mission to MARS Rover's embedded system needs to be able to function under extreme climate and unexpected environmental conditions [32]. These types of embedded systems require special environment monitor routines to maintain the embedded system's internal working environment.

2.2 Overview of artificial intelligence systems

The field of artificial intelligence has been around since the late 1940s [33]. The AI field refers to the study of implementing intelligent machines to do tasks that require human intelligence. According to the Oxford dictionary, the definition of the word "intelligent" is to show a high degree of understanding, and to be quick to comprehend [34]. Therefore, intelligent machines are machines that have the ability to understand and comprehend things in order to perform their tasks. To date, there are no intelligent machines that can actually understand and comprehend their tasks, but instead they exhibit behaviours that are similar to those an intelligent human would exhibit. An example of this is Deep Blue, the chess playing supercomputer [35]. Although in May 1997, Deep Blue defeated the world chess champion Garry Kasparov, the level of intelligence in Deep Blue is still considered to be very low. The reason Deep Blue

managed to defeat Garry Kasparov was because Deep Blue was able to search through 200 million chess moves per second and not because it is intelligent.

In order for these machines to exhibit intelligent behaviour, they use various AI techniques that have been developed throughout the years. Artificial intelligence systems, or intelligent systems for short, are machines that use AI techniques to exhibit intelligent behaviour. There have been some limited successes of intelligent systems, for example intelligent chess playing machines [36], intelligent medical diagnostic systems (MYCIN) [37] and intelligent automatic vacuum cleaners (Trilobite) [38][39].

AI techniques can be broadly broken down into two classes: knowledge-based system techniques and computational intelligence techniques [5]. Knowledge-based system techniques involve explicitly representing knowledge in the form of rules or other symbolic representations in the machine. Data is then manipulated with these rules to form the solution to the AI problem. Examples of knowledge-based systems are expert systems and agent systems. Computational intelligence techniques involve representing the AI problem in numerical form. The conclusion to the AI problem is then computed by performing arithmetic operations on the numbers. Examples of computational intelligence techniques are evolutionary algorithms and artificial neural networks. There are also some hybrid systems that use both knowledge-based system techniques and computational intelligence techniques. Examples of these are fuzzy logic systems and Bayesian theorem systems.

The reason there are so many different AI techniques available is because no single AI technique is suitable for solving all AI problems. Each technique is suitable for solving

particular types of AI problems. Usually, for complex real world problems, a mixture of AI techniques is required. For example in [40], a mixture of fuzzy logic and neural networks is used to drive a simulated racing car. The results show that the car's performance is better using a combination of fuzzy logic and neural networks than using fuzzy logic or neural networks alone. Another example of mixed AI techniques can be found in [41], where an attempt at generating creative musical rhythm is made by using a neural network to calculate the fitness value of the genetic algorithm (a type of evolutionary algorithm).

One way to integrate all these different AI techniques together is to use a blackboard architecture. The blackboard architecture consist of different 'experts' working together to solve a problem. These different 'experts' can represent the different AI techniques that can be used. Multi-agent systems are the latest field of research that can also integrate different intelligence into different agents and having them work together to solve an overall problem. This is similar to the blackboard architecture except that the agents can communicate directly with each other while in the blackboard architecture, the 'experts' can only communicate with each other via the blackboard. In a sense, blackboard systems can be considered as a subset of multi-agent systems. In fact, blackboard architectures have been used frequently as the implementation architecture for multi-agent systems [42][43]. Multi-agent systems are a broad research area and because the focus of this thesis is on blackboard systems, the following sections will only discuss blackboard systems in more detail. Further information on multi-agent systems can be found, for example, in [44][45].

2.2.1 Blackboard systems

A blackboard system is a software architecture that is based on the analogy of a group of experts working together to solve a common problem by writing their ideas onto a common blackboard. Figure 4 shows the analogy of a blackboard system with the blackboard as the common database.

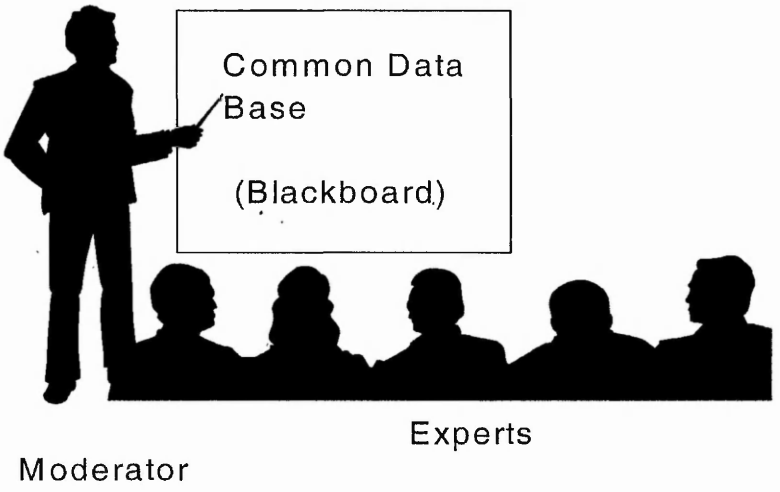


Figure 4. Analogy of a blackboard system

Figure 5 shows how the blackboard analogy is represented in software. There are two main classes of blackboard systems, traditional non-distributed blackboard systems and distributed blackboard systems.

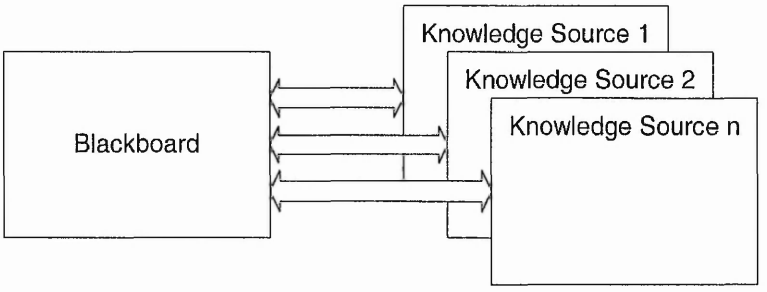


Figure 5. Software model of a blackboard system

2.2.1.1 Non-distributed blackboard systems

The traditional non-distributed blackboard system consists of three main modules: the blackboard module (BB), the expert or knowledge source module (KS), and the control module (also known as the scheduler module). The BB module is a global shared memory area where KSs can read and write information. A KS can be any software module such as rule-based production systems, neural networks, fuzzy logic systems, genetic algorithms, or just procedural algorithms. The system's current state of understanding of a problem is stored on the BB as it develops from a set of data towards a conclusion. The sets of data on the BB are organised into different levels/partitions and each KS would work on different levels/partitions of the BB.

The first blackboard system was developed around 1976 at the Carnegie-Mellon University in the Hearsay-II speech understanding project [46]. Later, the original developers of Hearsay-II moved on to different universities and started their own versions of blackboard systems that concentrated on different aspects of blackboard systems. The blackboard framework was then picked up by other researchers and was modified to suit the different researcher's needs. This has led to the usage of blackboard systems in a variety of applications. More recent applications of the blackboard architecture include: control of a mobile platform (1992) [47], schema integration (1995) [48], fault diagnosis on low voltage distribution networks (1998) [49], and medical diagnosis (2000) [50].

The Open University's Intelligent Computer Systems Research Group has developed a non-distributed blackboard system called ARBS (Algorithmic and Rule-based Blackboard System). ARBS has been successfully applied to the interpretation of ultrasonic images [51], the management of a telecommunications network [52], and the

control of plasma deposition processes [53]. Another successful non-distributed blackboard system called BEST (Blackboard-based Expert System Toolkit) was developed at the Mihailo Pupin Institution, Belgrade [54]. BEST has been successfully applied in a diagnostic system for the aluminium industry [55] and in an investment advisory system (INVEX) [56].

All these non-distributed blackboard systems include a controller module to control the execution of the KSs. This makes the blackboard system not truly opportunistic as the KSs cannot all execute whenever they want to but instead they have to wait for the controller to schedule the KS for execution. Fathi has argued that having a controller module is advantageous as it allows different scheduling or control strategies to be explored [57] but this is only useful if the domain knowledge of the problem is well understood. The control strategy can also be opportunistic, but in general there is an overhead required for evaluating the control strategy at every turn to decide which KS to execute next. One way of avoiding the use of a controller module is to have a distributed blackboard system with all the KSs running in parallel on separate processors.

2.2.1.2 Distributed blackboard systems

There are two main research fields in distributed blackboard systems. One is the same as the traditional non-distributed blackboard system except that the KSs and the BB are run on separate processors (in parallel) or as separate threads on the same processor (concurrently) [58][59]. The other is where many mini blackboard systems are networked together to form a large blackboard system [60][61]. Both of these systems can be run on a single processor with multiple threads/processes running concurrently or on different processors running in parallel.

An example of the second type of distributed blackboard systems (network of mini blackboard systems) can be seen in Naaman et al's fractal blackboard framework [61]. Here the KS of the upper-level blackboard system is the blackboard system of a lower-level. The KS of this lower-level blackboard system is in turn the blackboard system of another lower-level blackboard system. This is repeated until the problem is broken down into a simple enough sub-problem that can be solved by a KS. The structure of a fractal blackboard system can be seen in Figure 6.

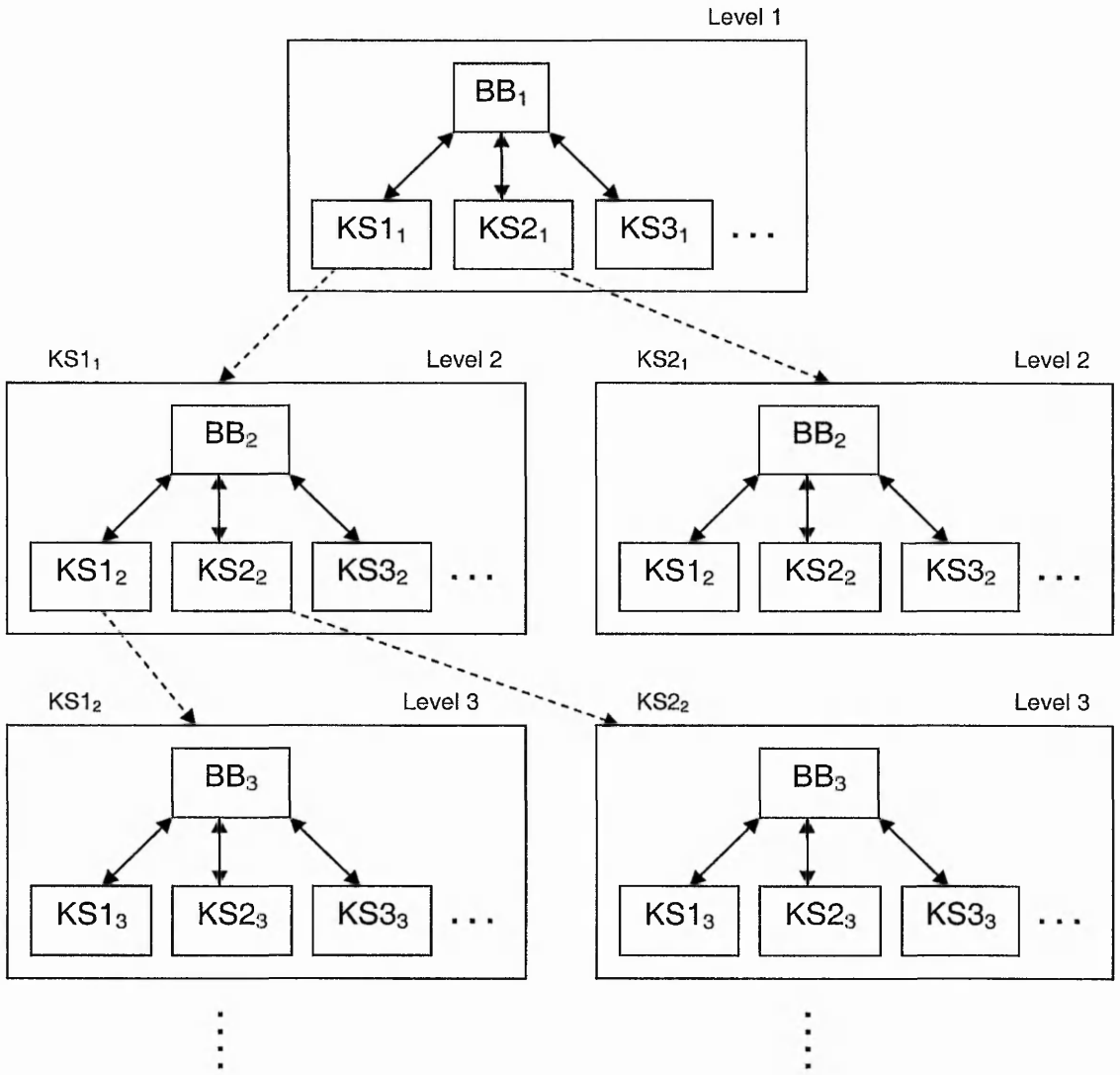


Figure 6. Structure of a fractal blackboard

Another example of a network of mini blackboard systems can be seen in Knowledge Technologies International's NetGBB [60]. Here, NetGBB makes use of multiple processors to implement the individual mini blackboard system. Each mini blackboard system is implemented in a processor (NetGBB node) and they are networked together via the NetGBB Interface. The network of mini blackboard systems still uses a controller module which means that it is essentially a traditional non-distributed blackboard system but with the BB or KS connected to other BBs or KSs. NetGBB was successfully implemented as the integration framework for Ford Research Laboratory's RRM (Rapid Response Manufacturing) Engineering Environment [62].

Another type of distributed blackboard systems is one that does not have a controller module. An example of this is DARBS (Distributed Algorithmic and Rule-based Blackboard System) developed by the Open University and Nottingham Trent University [59]. In DARBS there is no controller module, instead the BB and all the KSs are run as processes of their own. This makes the blackboard system closer to its analogy of a group of experts who communicate their ideas that help towards solving a problem through the BB. This means that the distributed blackboard system without a controller module is more truly opportunistic than the traditional non-distributed blackboard system with a controller module.

Blackboard systems have been argued to be similar to a multi-agent system but Ferber [45] agues that blackboard systems are not truly multi-agent systems as they have a centralised controller module. Distributed blackboard systems on the other hand better resemble multi-agent systems as each KS runs as an independent process on its own

without the need for a controller module. Each KS represents an agent that communicates with other KS agents via the BB. The BB itself can also be argued to be an agent that provides services to the other KS agents, e.g. communication services, storage space services and storage space update services.

2.3 Overview of intelligent embedded systems

Current day embedded systems are generally unreliable when their operating condition falls outside their narrow design specifications [63] or when an unexpected condition occurs. Unfortunately, the real world environment constantly changes and unexpected events occasionally occur. As biological beings, humans have developed adaptability skills to cope with this. The same should apply for future embedded systems. In order for embedded systems to be more reliable, they must be able to adapt to their changing environment. To be able to adapt, they must have some form of intelligence. Therefore, intelligent embedded systems are the next evolutionary step for embedded systems. Intelligent embedded systems are embedded systems with some form of artificial intelligence software incorporated into them. A well known example of an intelligent embedded system is Sony's robot dog, AIBO [64]. AIBO can learn and adapt to its owner and environment by recognising faces and objects. Electrolux's Trilobite [38][39] is another example of an intelligent embedded system. Trilobite is an intelligent vacuum cleaner that can adapt to its environment (in this case, the room it is in) and automatically vacuum the entire room.

A general block diagram of an intelligent embedded system is the same as an embedded system in Figure 3 except that the software now contains some form of artificial intelligence software. Like all embedded systems, complex intelligent embedded systems

can be made up of many smaller intelligent embedded systems connected together. Using the previous car example, the modern intelligent car contains small modules of intelligent embedded systems. For example, adaptive cruise control [65], rain-sensitive windscreen wipers and light-sensitive headlights. The adaptive cruise control uses fuzzy logic to adapt the cruise speed to the speed of the car in front and to the driver's needs, i.e. sensed from the accelerator pedal [65]. The rain-sensitive windscreen wipers can sense the amount of water on the windscreen and automatically adjust the wiper's speed accordingly. Finally, light-sensitive headlights can sense the amount of light in the environment and automatically switch on the headlights when it gets too dark.

Presently, intelligent embedded systems exhibit relatively simple intelligent behaviours. To exhibit more sophisticated intelligent behaviours, complex artificial intelligence software is required. For real-time embedded systems, running complex artificial intelligence software can be a problem. Complex artificial intelligence software usually takes an unpredictable amount of time to reason before it can respond to the changes in the environment. This unpredictable time for reasoning is not acceptable for real-time embedded system. A late intelligent response is just as bad as a wrong response. Generally, the more intelligent the response the longer the processing time is. Therefore a balance between how intelligent the response is and the amount of time required for processing the response needs to be struck. The field of real-time artificial intelligence (RTAI) delves into the research of implementing artificial intelligence software that has time deadlines [66].

Currently, there are limited successes in implementing real-time intelligent embedded systems. Most of the intelligent embedded systems have limited processing time and

power which severely limits the level of intelligence of the system. An example of real-time artificial intelligence system is in OASIS [67]. OASIS is a real-time air traffic management system that can handle realistic peak hour traffic samples from Sydney airport. OASIS is fairly intelligent but it demands powerful UNIX workstations. Another example of a real-time intelligent embedded system is in University of Michigan's Uninhabited Aerial Vehicle (UAV) project [68][69]. They have developed CIRCA-II, a Cooperative Intelligent Real-time Control Architecture for their UAV. The intelligence on the CIRCA-II is fairly low as it still requires a human pilot backup. Currently, a lot more intelligence is required for their UAV in order for it to achieve fully autonomous flight [68].

The constraints of an intelligent embedded system are the same as for an embedded system but with the added exception that it requires more processing power. This is to accommodate the extra processing power required by the artificial intelligence software. In the future as the requirements of the users become more sophisticated, more complex intelligent behaviour will be needed. This is where the extra processing power is required. There are two ways of increasing the processing power of the embedded system, one is to use a high MIPS processor and the other one is to have many lower MIPS processors connected together. There is a physical speed of light limit [11] for increasing a processor's MIPS. Using a distributed processing network is an alternative way to further increase the processing power. A general block diagram of an intelligent embedded system with these distributed processors is shown in Figure 7.

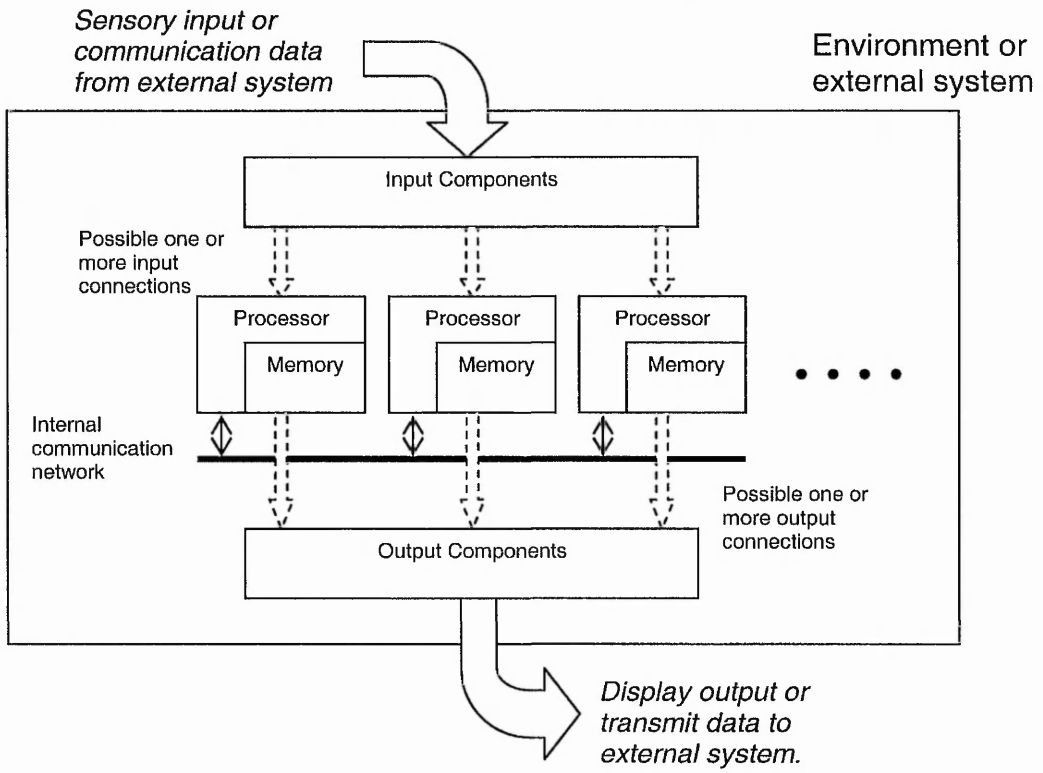


Figure 7. General block diagram of an intelligent embedded system with distributed processors

2.4 Overview of distributed processing networks

Distributed processing networks are a subset of the parallel processing architecture. Flynn's taxonomy of parallel architectures is based on the instruction and data elements of a system [70]. This is broken up into Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD) as shown in Figure 8.

Data / Instructions	Single	Multiple
Single	SISD	SIMD
Multiple	MISD	MIMD

Figure 8. Flynn's taxonomy

Flynn's taxonomy is well accepted in the parallel processing world, but fails to classify modern computers. This has led to various modifications to Flynn's taxonomy, one of which is Duncan's taxonomy [71]. Figure 9 shows Duncan's modification to Flynn's taxonomy and it can be seen that distributed processing networks (also known as distributed memory) belong to the MIMD model.

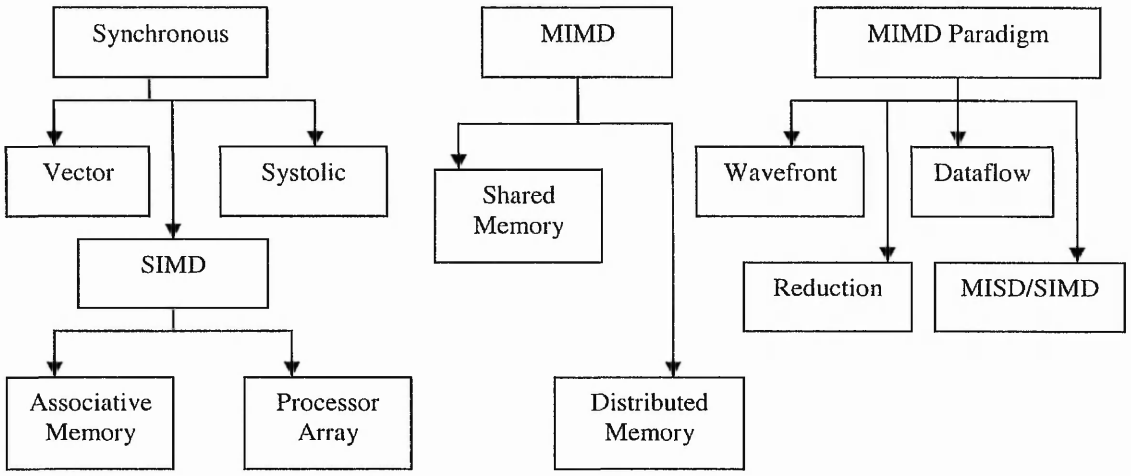


Figure 9. Duncan's Taxonomy

One of the major studies of distributed processing networks is in interconnection network topologies. There are many different network topologies being used in distributed processing networks such as linear networks, ring networks, star networks, hypercube networks and fully connected networks. The more interesting network topology is the hypercube network and examples of this network include the Intel iPSC [72] and nCUBE [73]. Most of the work using the Intel iPSC is on intensive numerical operations. For example, using iPSC to perform distributed simulation of timed Petri nets [74]. Unfortunately, Intel has ceased production of the iPSC systems [75]. nCUBE,

on the other hand, has been successfully implemented in on-demand video streaming systems [73].

However, one problem with the hypercube network topology is that communication between processors that are more than one processor away must pass through intermediate processors. If there is heavy communication through a processor, then that processor might be delayed in receiving its own messages. Ideally, the communication topology should be point-to-point (fully connected network) but this would be too costly or impractical for a large network of processors.

One approach to interconnection is to have a router or switch network topology. This is also known as a dynamic interconnection network [76] as the connection to each processor is determined dynamically at runtime. The IBM SP-2 uses this type of network topology and has been used with great success for linear feature extraction from images of up to 512 × 512 pixels [77]. Another distributed processing network that can use the switch network topology is the INMOS transputer [78]. Research from the University of Edinburgh that uses the Meiko multi-transputer system in a torus network topology showed that the torus topology requires careful planning to distribute the processes in the network to minimise communication time [6]. This has led to the development of routers or switchers to offload the communication burden from the transputers. One such router is the ICR C416 router developed by ICRouting Ltd [79].

Unfortunately, INMOS was sold to ST Microelectronics in 1986 and ST Microelectronics has ceased production of the transputers. The Parallel Processing Research Group from Nottingham Trent University then developed the SARNet to

replace the diminishing transputers [80]. SARNet consist of a switched network of StrongARM RISC processor nodes (SARNodes) [81]. Each SARNode consists of a 32-bit 200MHz StrongARM SA-110 RISC processor, 8 megabytes of SDRAM, a 32-bit 1MHz real-time timer, a UART debug port, and an OS-Link (over sampling link) [79] communication module. The switcher or router used is the same as the ones originally used for transputers (i.e. the ICR C416). A typical network layout of the SARNet distributed processing network is as shown in Figure 10.

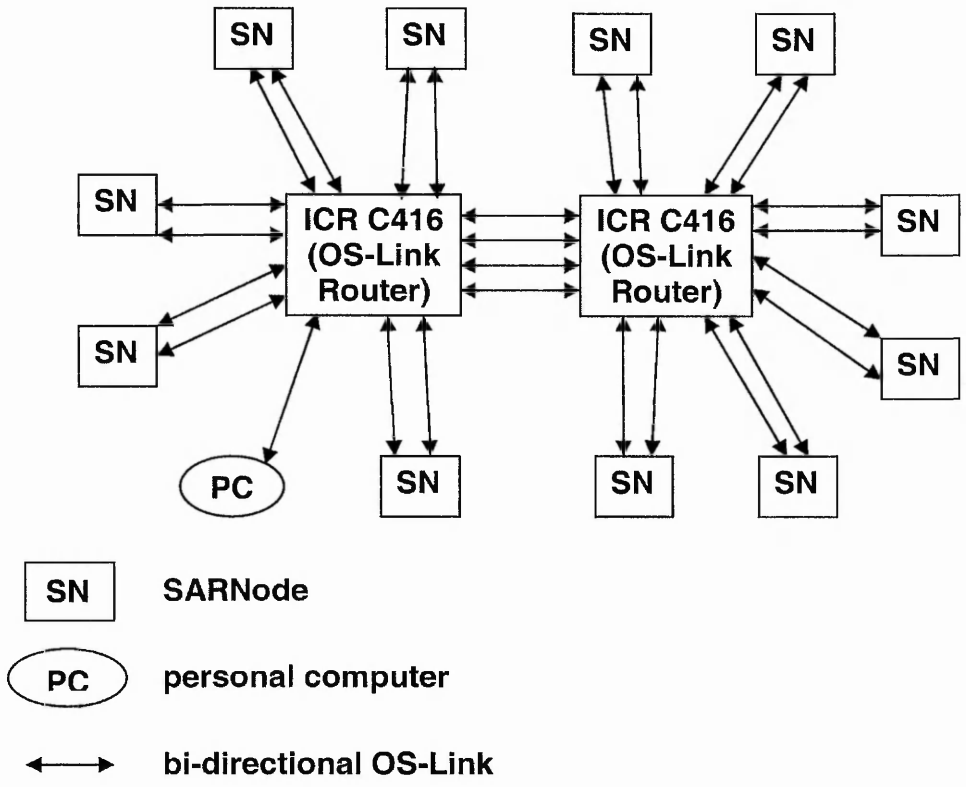


Figure 10. Typical network topology of SARNet

2.5 Overview of distributed blackboard systems in distributed processing networks

There are a few distributed blackboard systems that have been implemented in distributed processing networks before. One such system is the distributed blackboard system in MACE (Multi-Agent Computing Environment) [82]. This system was

implemented on a 16-node Intel Sym-1 large-memory hypercube environment. Another blackboard system implemented on the hypercube topology is the tactical decision generator on the CUBE CLAWS system [58]. As mentioned earlier, the disadvantages of a hypercube network is in its communication bottleneck when the communication volume is high. Another system with a different network topology is the vision application program implemented in a torus network on the Meiko multi-transputer system [6]. This implementation showed relatively good performance but careful distribution planning was required.

A slightly different type of distributed processing networks that has been used for the blackboard system is that of using the Internet as the communication network. An example of such a system is the cell-based design support system for composite structures [83]. Here the local blackboard system that supports the cell-based design can consult the database of another blackboard cell-based design support system to find a better composite structure that meets its current requirements. A more recent and interesting work is in implementing distributed blackboard system in mobile robots [84]. Here, the mobile robots are scattered all over a room and their job is to sense their surrounding environment and build up a detailed map of the room on the BB. The mobile robots are the KSs (called agents in this paper) and they are connected to their central BB via wireless Ethernet and radio modem connection. The overall aim was to be able to have strategic-level path-planning from the detailed map of the room, but currently only the map-building part has been accomplished.

Apart from DARBS [59], all distributed blackboard systems reported in the literature have a control module. The current implementation of DARBS uses the popular TCP/IP

communication protocol which is supported by the Ethernet switch network topology. As mentioned in section 2.4, the router or switch network topology is a good topology as it is comparable to point-to-point connection. Therefore, DARBS is a suitable candidate for testing the performance of distributed blackboard systems.

2.6 Summary

The main constraints of embedded systems are cost, processing power, memory capacity, reliability, electrical power consumption, physical size, real-time, and working environment. Different embedded systems have different degrees of these constraints to meet. Future embedded systems need to be able to adapt to the changing environment and needs of the user. To do this, complex artificial intelligence software, which is processor intensive, is required. Therefore, there is a need for artificial intelligence software that is not processor intensive yet complex enough to be able to learn and adapt to the user's needs.

The field of artificial intelligence has been around since the late 1940s. To date, there are no real intelligent machines around but instead there are machines that exhibit intelligent behaviours. There are many different artificial intelligence techniques available, for example, rule-based systems, neural networks, genetic algorithms, and fuzzy logic. No single artificial intelligence technique is suitable to solve all types of problems. One way to integrate different artificial intelligence techniques together is to use the blackboard architecture. The blackboard architecture is based on the analogy of a group of experts working together to solve a common problem by writing their ideas onto a common blackboard. There are two main classes of blackboard architecture, distributed blackboard systems and non-distributed blackboard systems.

As future embedded systems need to be able to adapt to their ever changing environment, intelligent embedded systems are the next evolutionary step for embedded systems. Intelligent embedded systems are embedded systems with some form of artificial intelligence software. Presently, intelligent embedded systems have relatively primitive intelligent behaviours. Complex artificial intelligence software is required for more sophisticated intelligent behaviours. For real-time embedded systems, complex artificial intelligence software causes real-time issues. The field of real-time artificial intelligence (RTAI) delves into the research of these issues. Complex artificial intelligence software requires high processing power. One way of increasing the processing power is to have distributed intelligent embedded systems.

Distributed processing networks are a subset of the parallel processing architecture. According to Duncan's taxonomy, which is based on Flynn's taxonomy, distributed processing networks (also known as distributed memory) belong to the Multi Instructions Multi Data (MIMD) model. One of the major studies of distributed processing networks is in interconnection network topologies. There exist many different network topologies for distributed processing networks. One network topology which is close to the ideal point-to-point network topology is the switch or router network topology. The Parallel Processing Research Group from Nottingham Trent University has developed a distributed processing network called SARNet which uses ICR C416 router.

There are a few distributed blackboard systems implemented on distributed processing networks. Each implementation uses different network topology. Presently, apart from DARBS, no other published distributed blackboard system uses a switch network topology without a control module. Therefore, DARBS is a suitable candidate for testing the performance of distributed blackboard systems. The next chapter will describe the implementation of a test application on a distributed blackboard system for later performance experiments.

3. Implementing a test application on a distributed blackboard system

This chapter explains in detail the aims and implementation of a test application on a distributed blackboard system. This implementation will provide the foundation for the experiments carried out in chapter 4.

3.1 Aims and requirements

One of the aims of this work is to investigate the performance of a distributed blackboard system in a distributed processing network. In order to do this, a suitable distributed blackboard system and a test application to run on it needed to be chosen. The distributed blackboard system chosen did not have a centralised control module. This was to make sure that the distributed blackboard system was truly opportunistic. The choice of distributed blackboard system was also based on the programming language used in order to make sure that it was portable, suitable for an embedded system and suitable for further development work to be done on it.

The choice of application to run on the distributed blackboard system was not important just so long as it met the basic requirements. The application did not have to be a practical application but had to be a naturally distributed problem that was suitable for a distributed blackboard system. It also had to be scalable so that performance experiments could be carried out on it. A well known application was appropriate as peers would be familiar with it and therefore, find it easier to understand. The application had to have the possibility for different KS implementations so that the advantages of distributed

blackboard systems could be fully utilised. This would also provide opportunity for future experiments involving different KS implementations.

3.1.1 Selecting a distributed blackboard system

At the onset of this project, there were commercially available distributed blackboard systems: one such system was Knowledge Technology International's NetGBB [60] which was an extension to their Generic Blackboard (GBB) product. However, problems with commercial products include cost and the fact that source code was not normally accessible. NetGBB was written in LISP and this is not a suitable language for embedded systems as embedded systems are typically prone to memory and speed constraints (section 2.1). LISP has been proven to be too slow to run on speed critical systems such as Pilot's Associate control-advisory system [85]. There are at the moment no commercial distributed blackboard systems that cater for embedded systems. There is a research-based blackboard system called BEST from Mihailo Pupin Institution, Belgrade [54] but that is a traditional blackboard system and full access to the source code was not possible.

Therefore, the distributed blackboard system that was chosen for this project was DARBS [59]. One of the reasons for choosing this distributed blackboard system was because it did not have a control module. This was one of the requirements and it would be an interesting opportunity to see how a distributed blackboard system without a control module performs in a distributed processing network. The other reason was that DARBS had been developed into a relatively stable working system and the source code for it had been provided freely with the intention that DARBS could be further improved during the course of this research. This meant that changes to the DARBS source code

could be done freely to suit the research needs. This was an essential requirement for the success of this research.

The source code for DARBS was written in C++ and this is a suitable language for embedded systems. Full access to the source code was also crucial to allow DARBS to be fully understood. This would then enable successful implementation of DARBS in the distributed processing network. The current DARBS source code was not efficient in terms of its coding for speed. It was therefore hoped that further parallelism could be achieved during the course of this project. Unfortunately, parallelism led to data consistency problems but these were addressed in this research.

3.1.2 Selecting an application

The choice of application to run on the distributed blackboard system was not important just as long as it allowed the performance experiments to be carried out. Creating a custom application for this purpose would be the easiest to implement but this would have made it difficult for peers to compare results as it would not be an established application. The MACE system in the University of Southern California used a simple arithmetic calculator application for their distributed blackboard system [82]. The KS in this arithmetic calculator represents each of the arithmetic operators, i.e. PLUS-KS, MINUS-KS, TIMES-KS and DIVIDE-KS. This application was deemed not suitable as it was too simple and not scalable beyond the four arithmetic operators. The University of South Carolina on the other hand, used chemical structure analysis as the application for their blackboard system but this was deemed to be too specific and difficult to understand [8]. It was also nearly impossible to implement the same application without an in-depth knowledge of chemical structure analysis.

Occello and Demazeau proposed using blackboard systems for implementing multi-agent systems [12]. Corkill has also suggested that the future of collaborating software consist of a hybrid of blackboard systems and multi-agent systems [86]. Therefore, a multi-agent system application was a suitable application to run on the distributed blackboard system. There were a few multi-agent system test-beds currently being used by multi-agent system researchers, one of which is TileWorld [87]. The TileWorld test-bed was deemed a suitable application to run on the distributed blackboard system because it is a naturally distributed problem and is scalable. The TileWorld test-bed is also a well established test-bed and, therefore, peers would be familiar with it. By implementing the individual agents in the TileWorld test-bed as individual KSs, different agent implementations were also possible (e.g. rule-based agents, neural network agents, fuzzy logic agents, etc.). The following section gives a more detailed overview of the TileWorld test-bed.

3.1.2.1 TileWorld test-bed

The TileWorld test-bed is a well established test-bed for multi-agent systems. The original TileWorld introduced by Pollack and Ringuette was a single agent TileWorld test-bed [87]. Later, Ephrati et al. introduced a multi-agent TileWorld called MA-TileWorld which is what multi-agent systems researchers now use as their TileWorld test-bed [88]. From here onwards, MA-TileWorld will simply be referred to as TileWorld. The TileWorld test-bed is a two dimensional grid world with objects in it. There are agents, holes, obstacles and tiles in the TileWorld. The objective of the agents is to score as many points as possible. They score points by moving around the TileWorld to find and pick up tiles which they then put into holes. The agents have a

limited view of the TileWorld. The viewing radius is a variable that can be set by the designer of the TileWorld. Each cell can only be occupied by one agent at a time. Agents cannot move to a cell with an obstacle in it. An example of a 10 × 10 TileWorld is shown in Figure 11.

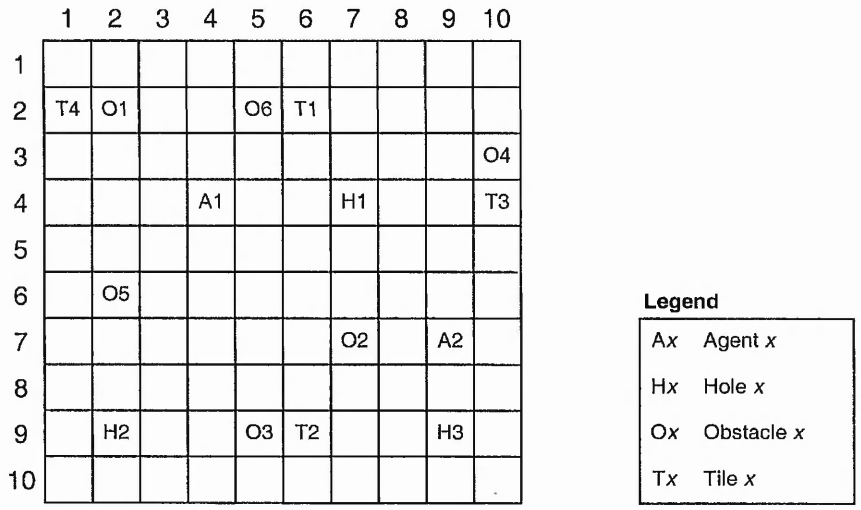


Figure 11. A 10 × 10 TileWorld

The holes, obstacles and tiles in the TileWorld can change dynamically, i.e. they can appear and disappear at different locations in the TileWorld. The rate of change is set by a variable, and this can be used to reflect the dynamically changing real-world environment. The TileWorld test-bed has been widely used to test the behaviour and interaction of multiple agents in a dynamic environment [89][90]. There is also a variant of the original TileWorld test-bed that includes "gas station" objects to top up the resources of the agents [91]. In this variant, the agent's resource-management skill is investigated by making each move consume fuel. Carrying a tile would cause the agent to consume more fuel. Therefore the agent would need to balance the consumption of resources with scoring points. However, for the sake of simplicity and due to the time limitation of this research, this variant was not used.

3.2 Design

This section will discuss the design of the TileWorld test-bed on the distributed blackboard system, DARBS. A detailed technical background on DARBS must first be explained before the actual design of TileWorld on DARBS can be understood.

3.2.1 Technical background on DARBS

DARBS (Distributed Algorithmic and Rule-based Blackboard System) [59] is the distributed version of ARBS (Algorithmic and Rule-based Blackboard System) [53]. DARBS, being a distributed blackboard system, has all its KSs and BB running as separate processes. The general structure of DARBS can be seen in Figure 12.

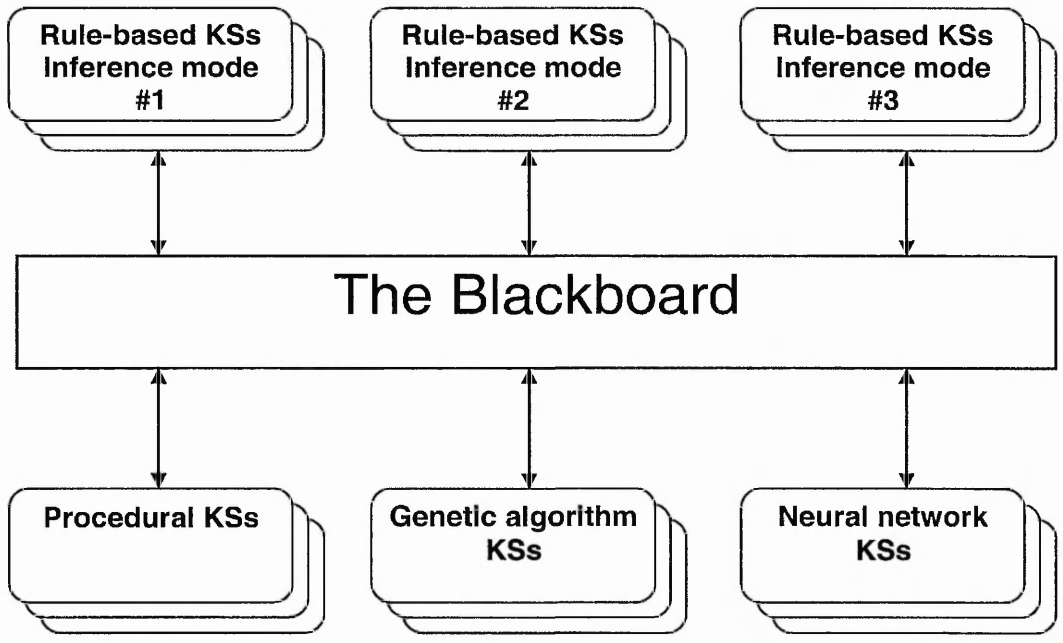


Figure 12. General DARBS structure

The main parts of DARBS that were important to this research are: DARBS data structure and command statements on the BB, and DARBS classes and Inter-Process Communication (IPC) model.

3.2.1.1 Data structure and command statements on the blackboard

Information on the BB is organised into partitions. The order in which the information is organised is entirely up to the designer. However, poor organisation of the information will slow down the BB's search for particular information. For example, putting all the information in one partition would make it slow to search for particular information in the partition. On the other hand, having too many partitions will slow down the search for particular partitions. Therefore, a balance between the number of partitions and the amount of information stored in the partitions was required.

DARBS's BB stores both the partitions and the information in the partitions in the form of strings, i.e. a list of characters. The naming of partitions and the information structure in each partition are entirely up to the designer. Ideally, the information stored on the BB should be as intelligible and informative as possible. This would make it easier for other KS designers to develop newer KSs for the problem on BB. Intelligible and informative information would also aid in debugging of the system. However, putting too much information on the BB will increase the communication overhead (i.e. longer messages take longer time to be sent to and from the BB). An example of how information is stored on the BB is as shown in Figure 13.

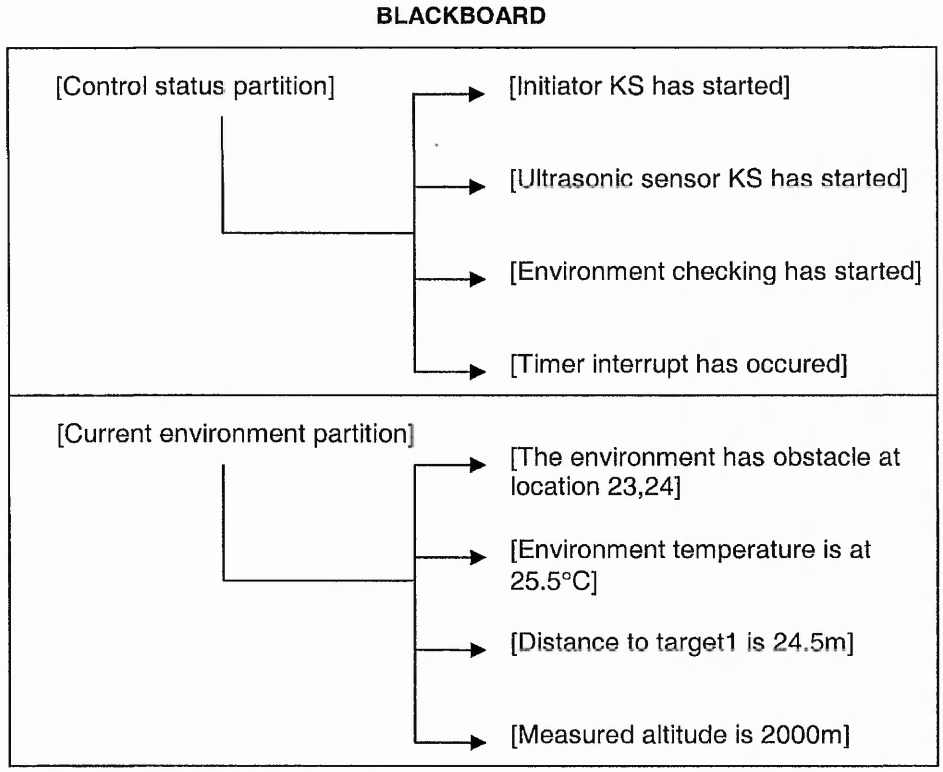


Figure 13. Example of data structure on DARBS's blackboard

DARBS command statements inherits most of ARBS command statements. The general command statement for DARBS is as follows:

```
DARBS_command [content] [partition]
```

An example of adding information to a partition is as follows:

```
add [Obstacle is at location 23,40] [Current Environment]
```

The BB would create a new partition called `[Current Environment]` if it does not already exist and then put the information `[Obstacle is at location 23,40]` in it. The BB also has commands to add multiple contents to the same partition. For example:

```
add_multi [Obstacle is at location 2,41] [Target is at location 4,10] [Water is at location 11,2] [Current Environment]
```

This command will add three more contents to the `[Current Environment]` partition. To retrieve the information in a partition, the command to use and the reply from the BB is as follows:

```
get_contents [Current Environment]
BB reply: [Obstacle is at location 23,40][Obstacle is at location 2,41][Target is at location 4,10][Water is at location 11,2]
```

The information on the BB can also be queried with `ret_all` (return all) command to match certain patterns, for example:

```
ret_all [? is at location 23,40] [Current Environment]
BB reply: true [ [ Obstacle ] ]
```

The BB replies `true` (as there is a match for that pattern) followed by the word that matches that string. The following command would return false:

```
ret_all [? is at location 40,10] [Current Environment]
BB reply: false
```

The `?` symbol matches a word and the `??` symbol matches one or more words in that position of the string. For example:

```
ret_all [Target is at ?] [Current Environment]
BB reply: false
ret_all [Target is at ??] [Current Environment]
BB reply: true [ [ [ location 4,10 ] ] ]
ret_all [Target is at location ?] [Current Environment]
BB reply: true [ [ 4,10 ] ]
```

The `==` symbol functions the same as `??` except that it does not return the matched words. This is useful for "don't care" conditions. For example:

```
ret_all [== is at location ?] [Current Environment]
BB reply: true [ [ 23,40 ] [ 2,41 ] [ 4,10 ] [ 11,2 ] ]
```

The full list of DARBS commands and an explanation of how to use them are in the DARBSCmd.h header file of the source code. A section of this header file can be seen in Appendix B. The full source code for the TileWorld implementation on DARBS is supplied in the CD-ROM attached to the back of this thesis in Appendix J.

3.2.1.2 DARBS classes and Inter-Process Communication (IPC) model

The design of DARBS is based on the client/server model where the BB is the server and the KSs are the clients. The KS clients are required to register themselves with the BB server by connecting to it. The KS clients can then send commands and wait for a reply from the BB server. When a KS client changes the contents of a partition, the BB server would broadcast a message informing the other registered KS clients that the partition has changed. It is then up to the individual KS clients to react to the changes on the partition.

The communication protocol that DARBS uses is TCP/IP. DARBS runs on a PC-based Linux operating system and each process can be run on a different PC that is networked together via TCP/IP communication protocol. Currently, DARBS has been implemented to solve the original ARBS problem of interpreting ultrasonic images of welds [51].

For each one of the DARBS processes (i.e. the BB server and each KS client) there is a communication module and a tokenizer module. The tokenizer module consists of only one class which is called `LnTokenizer`. This class is used to break down the message that is received into small tokens of one word each to be interpreted by the main program.

The communication module on the other hand consists of two classes: the base class `LnSignalHandler` and the derived class `LnTcpServer` (for the BB server) or `LnTcpClient` (for the KS client). A unified modelling language (UML) [92] diagram is shown in Figure 14 to illustrate these classes.

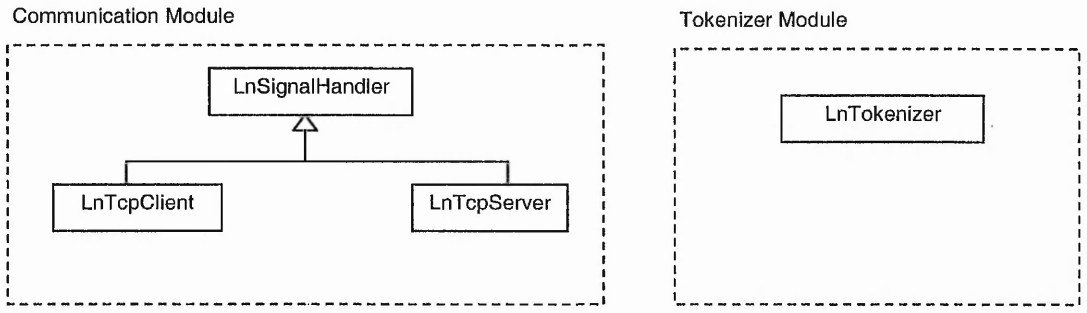


Figure 14. DARBS communication and tokenizer module

The `LnSignalHandler` class is a pure virtual class that is used to store all the pointers to the communication interrupt handlers (called signals in the UNIX environment) [93]. The `LnTcpServer` class is a derived class from the `LnSignalHandler` class that handles the server side of the TCP/IP communications. This class provides all the necessary functions for opening and closing a listening port, sending and broadcasting messages, and setting up the receiving message interrupt function. The `LnTcpClient` class is the client side equivalent of `LnTcpServer` class. The only difference being that `LnTcpClient` does not have the broadcast function and has a connect-to-server and close-connection-from-server instead of opening and closing listening port.

For the BB server, there is a `LnBlackboard` class that handles storage creation and deletion. It also manages the searching for data items on the BB. On the KS client side there are two main modules: the core KS client module and the graphical user interface

(GUI) module. Only the main classes of the most common KS client modules are explained here as a full description of all KS client modules will be too long for this thesis.

The main KS client module consists of the `pwCClient` class that handles the initialisation of the client, reading the KS file (to determine what type of KS this instance of the client is), setting up the connection to the BB server, creating the KS, and finally executing the KS. All this is done with the help of the `LnTcpClient` class and the `pwCKnowledgeSource` class. The `pwCKnowledgeSource` class is the base class for the different types of derived KS classes that are supported by DARBS. Currently, the supported KS classes are rule-based KS (`pwCRuleKS` class) and procedural KS (`pwCProceduralKS` class). Work is in progress to implement neural network KS (`pwCANNKS` class). Figure 15 illustrates the core KS client's classes in UML standard.

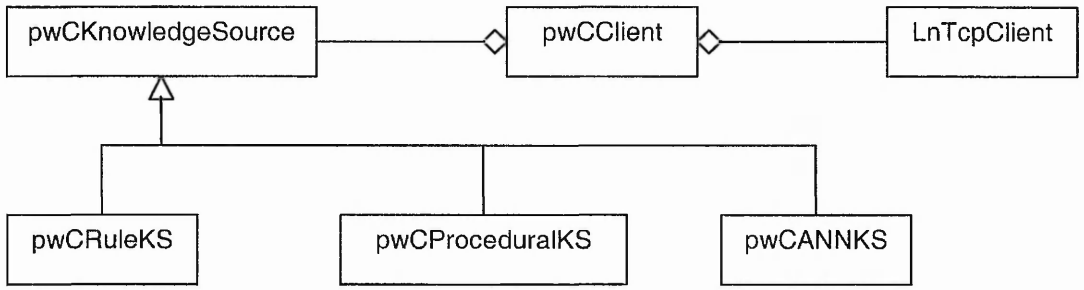


Figure 15. Core KS client classes

DARBS's communication module uses the Linux operating system's IPC model; therefore, Linux IPC needs to be explained in a bit more detail. The Linux operating system uses signals as a form of interrupts. Each process in the Linux operating system can register its interrupt service routine or function to be called when a particular interrupt/signal is generated [93]. For example, the signal that is generated when a

message is received on the Linux's IPC model is `SIGIO` [93]. So by registering a function that is triggered by a message from the communication channel to the `SIGIO` signal, would allow this function to be called whenever a message is received on the Linux's IPC model. Linux's IPC also provides a listen to port function for the server. Basically in the client/server model, the server initiates a service on a port and listens on that port for possible clients. Once the server has started to listen on a port, the client can make a call to connect to that port and it would be automatically passed to the server. The server can then register the client's file descriptor and start to provide services to it. Linux's IPC also provides a function call to send data to a process (client or server) by using the process's file descriptor. DARBS's `LnTcpServer` and `LnTcpClient` classes make use of these functions provided by Linux to implement the client/server model. `LnTcpServer` class has an extra broadcast function that sends a message to all the clients except for the current client (i.e. the client that last sent a message to the server).

3.2.2 Designing TileWorld on DARBS

The natural way the TileWorld fits in a blackboard system made it the ideal choice for implementing it in DARBS. The TileWorld environment is the environment where all the agents and other objects interact with each other. Therefore, this made it suitable to implement the agents as KS clients and storing information of the TileWorld environment on the BB. This would give all the agents in the TileWorld equal access to see and change the world. The set-up of TileWorld with five agents on DARBS is as shown in Figure 16.

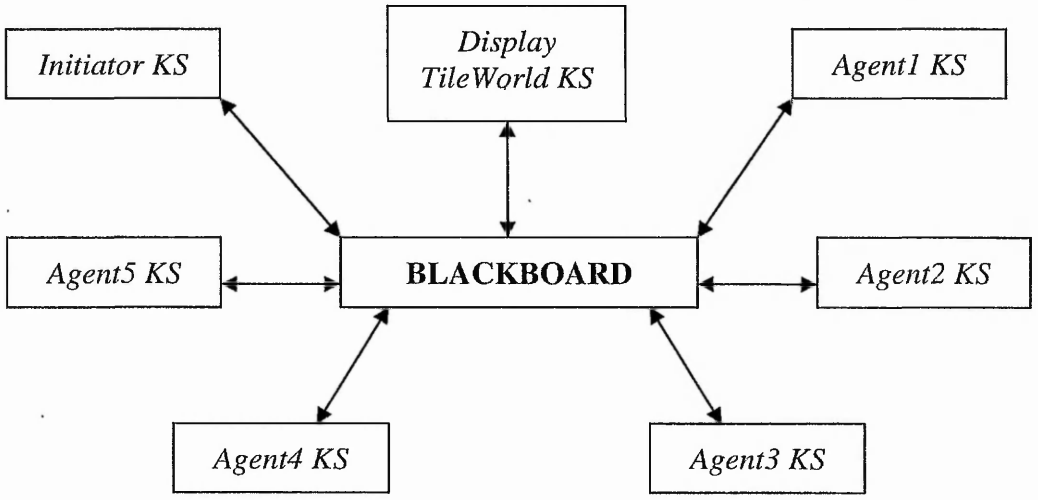


Figure 16. Five agents TileWorld set-up on DARBS

The *Initiator KS* is the KS that sets up the TileWorld with its parameters and generates the actual TileWorld. The *Initiator KS* is only needed in the beginning, after setting up the TileWorld environment, it terminates. The *Display TileWorld KS* is the KS that displays the TileWorld and its contents in a graphical format. This makes it easier for users to follow the changes on the TileWorld. To do this, the *Display TileWorld KS* needs to keep track of the changes on the TileWorld environment (stored on the BB) and make sure that the graphical representation of the TileWorld is as up-to-date as possible. The *Agent KS*s are the actual agents. They control their respective agents on the TileWorld and make changes to the TileWorld environment on the BB accordingly. For the sake of simplicity of design and time limitations, the objects in the TileWorld (i.e. tiles, holes, and obstacles) are all static and do not appear and disappear with time. However, it would be possible to make the objects in the TileWorld dynamic, i.e. appear and disappear with time. This could be done by simply adding another KS that uses a probability to change the number and position of the tiles, holes, and obstacles in the TileWorld.

Careful organisation of the information on the blackboard was required to make sure that the agents and all other KSs could interact with each other properly through the blackboard. The organisation of the information also needed to minimise the number of partitions that a particular KS was working with in order to reduce the number of times the KSs needed to restart. A KS would restart when a partition that it was working with had changed. The format of the information on the blackboard also needed to be carefully constructed so that queries from the KS clients were as easy and efficient as possible. Finally, the overall information organisation and format construction needed to be easily understandable in order to permit other users to easily understand and modify the TileWorld. With all these in mind, the following partitioning of the blackboard was designed (see Figure 17).

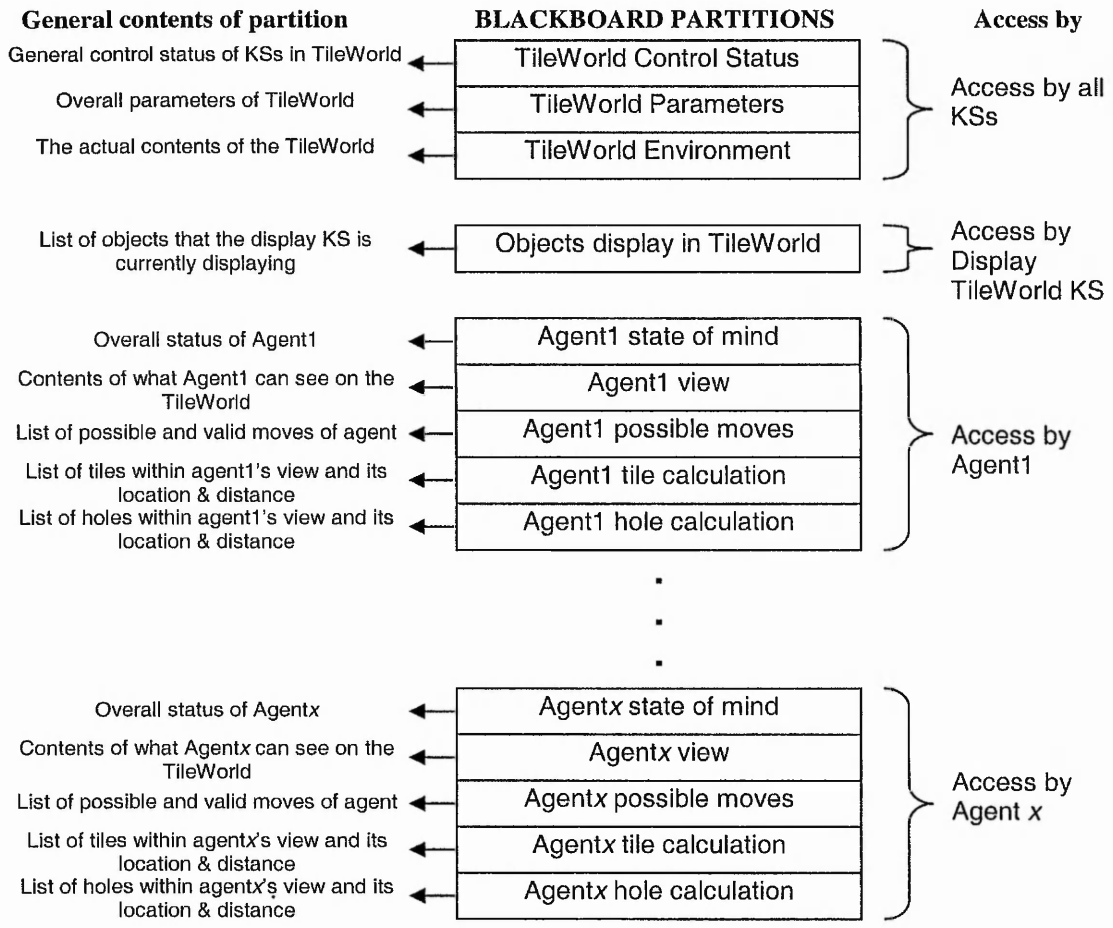


Figure 17. Partitions on blackboard, its general contents and KSs that access them

As can be seen in Figure 17, all *Agent KS*s can access the *TileWorld Environment* partition and, in theory, can see the whole TileWorld. This could be argued to be an incorrect implementation of the TileWorld test-bed, but it was assumed that all agents are benevolent and will only access those areas of the *TileWorld Environment* partition that they are intended to.

There are too many information strings on the blackboard to explain each in detail but the most important information string format is that on the *TileWorld Environment*

partition. The information string format on the *TileWorld Environment* partition is as follows:

`[Location x , y contains` *`NO AGENT , NO HOLE , NO OBSTACLE , NO TILE`*`]`

The order of objects is important as this allows easy query by the KSs. The order is Agent→Hole→Obstacle→Tile. If the location contains an agent then *Agent x* (where *x* is the agent number) replaces *NO AGENT* and the same goes for holes, obstacles and tiles. Examples of these are as follows:

```
[Location 1 , 4 contains Agent 3 , NO HOLE , NO OBSTACLE ,
NO TILE ]
[Location 1 , 5 contains NO AGENT , Hole 21 , NO OBSTACLE ,
NO TILE ]
[Location 1 , 6 contains NO AGENT , NO HOLE , Obstacle 18 ,
NO TILE ]
[Location 2 , 4 contains NO AGENT , NO HOLE , NO OBSTACLE ,
Tile 2 ]
[Location 10 , 2 contains Agent 5 , NO HOLE , NO OBSTACLE ,
Tile 4 ]
```

This information string format allows easy queries to be made to check if an agent, hole, obstacle or tile is present in a specific location in the TileWorld. An example to check if an agent is present in location 1 , 4 is as follows:

```
ret_all [Location 1 , 4 contains Agent ? , NO HOLE , NO
OBSTACLE , NO TILE] [TileWorld Environment]
```

The reply from the BB would be true if an agent exists in location 1 , 4 otherwise a false would be returned. In this example, the following reply could be received:

```
BB reply: true [ [ 3 ] ]
```

This reply means that there was an agent present in location 1 , 4 and that that agent is `Agent 3`. This information string format also allows searching for the location of a specific agent, hole, obstacle or tile. An example of searching for the location of `Obstacle 18` is as follows:

```
ret_all [Location ? , ? contains NO AGENT , NO HOLE ,
Obstacle 18 , NO TILE] [TileWorld Environment]
BB reply: true [ [ 1 6 ] ]
```

The reply here means that `Obstacle 18` was found in location 1 , 6. The full contents of a specific location could also be queried using the following example:

```
ret_all [Location 2 , 4 contains ??] [TileWorld Environment]
BB reply: true [ [ [ NO AGENT , NO HOLE , NO OBSTACLE ,
Tile 2 ] ] ]
```

For the *Display TileWorld KS*, an important search facility would be to search for the locations of all the agents, holes, obstacles, and tiles in the TileWorld. An example to search for the locations of all the agents is as follows:

```
ret_all [Location ? , ? contains Agent ? , == ] [TileWorld
Environment]
BB reply: true [ [ 1 4 3 ] [ 10 2 5 ] ]
```

The reply here means that `Agent 3` was at location 1 , 4 and `Agent 5` was at location 10 , 2. As can be seen here, this information string format has provided an intelligible, easy to query, and efficient way of storing the TileWorld environment information.

As the ultimate aim of this implementation was to run the TileWorld in a distributed processing network, there was a risk of data inconsistency when objects in the TileWorld

environment were moving from one location to another. To move objects from one location to another, the object must first be removed from one location and then added to another location. This could have been accomplished in four primitive DARBS commands, i.e.:

```
del_all [Location 1 , 4 contains Agent 3 , NO HOLE , NO
OBSTACLE , NO TILE] [TileWorld Environment]
add [Location 1 , 4 contains NO AGENT , NO HOLE , NO
OBSTACLE , NO TILE] [TileWorld Environment]
del_all [Location 2 , 4 contains NO AGENT , NO HOLE , NO
OBSTACLE ,Tile 2] [TileWorld Environment]
add [Location 2 , 4 contains Agent 3 , NO HOLE , NO
OBSTACLE , Tile 2] [TileWorld Environment]
```

However, during the time between these four instructions, other KSs may accidentally read the contents of the *TileWorld Environment* partition. This would cause the KSs to have inconsistent data as the data is currently changing. Therefore, three new DARBS commands were added to solve this problem: `add_multi`, `replace` and `replace_multi`. The `add_multi` command has already been explained in section 3.2.1.1. The `replace` command is a combination of delete and add commands. So the above instructions to move `Agent 3` to location 2 , 4 could be reduced to the following:

```
replace [Location 1 , 4 contains Agent 3 , NO HOLE , NO
OBSTACLE , NO TILE] [TileWorld Environment] [Location 1 , 4
contains NO AGENT , NO HOLE , NO OBSTACLE , NO TILE]
replace [Location 2 , 4 contains NO AGENT , NO HOLE , NO
OBSTACLE ,Tile 2] [TileWorld Environment] [Location 2 , 4
contains Agent 3 , NO HOLE , NO OBSTACLE , Tile 2]
```

This instruction is useful for replacing one item in a partition but moving agents in the TileWorld requires two `replace` commands. Therefore, the `replace_multi` command was introduced to do multiple replacements in one instruction. The following shows how `replace_multi` can be used to move `Agent 3` to location 2 , 4:

```
replace_multi [Location 1 , 4 contains Agent 3 , NO HOLE ,
NO OBSTACLE , NO TILE] [TileWorld Environment] [Location 1 ,
4 contains NO AGENT , NO HOLE , NO OBSTACLE , NO TILE]
[Location 2 , 4 contains NO AGENT , NO HOLE , NO
OBSTACLE ,Tile 2] [TileWorld Environment] [Location 2 , 4
contains Agent 3 , NO HOLE , NO OBSTACLE , Tile 2]
```

As can be seen here, the data inconsistency problem does not occur when using `replace_multi` command.

3.3 Implementation

There are three types of KSs in this TileWorld implementation. They are: *Initiator KS*, *Display TileWorld KS* and *Agent KS*. These KSs were all implemented as rule-based KSs and the following sections will explain their implementation in detail.

3.3.1 Initiator KS

The main purpose of the *Initiator KS* is to set up the parameters of the TileWorld and to generate the actual TileWorld from these parameters. These parameters are set in the rules of the KS and they include: the size of the TileWorld and the number of agents, holes, obstacles and tiles in the TileWorld. These parameters are stored on the *TileWorld Parameters* partition. Once these parameters have been set up on the BB, the *Initiator*

KS can create the TileWorld and randomly place the agents, holes, obstacles and tiles in the TileWorld. This was done by using the standard C++'s `rand()` [94] function with a user define random seed number. Once the TileWorld with all its objects has been created and stored on the BB, the *Initiator KS* will terminate. Figure 18 shows the flow chart of the *Initiator KS*. A detailed list of *Initiator KS* rules and their functions can be seen in Appendix C.

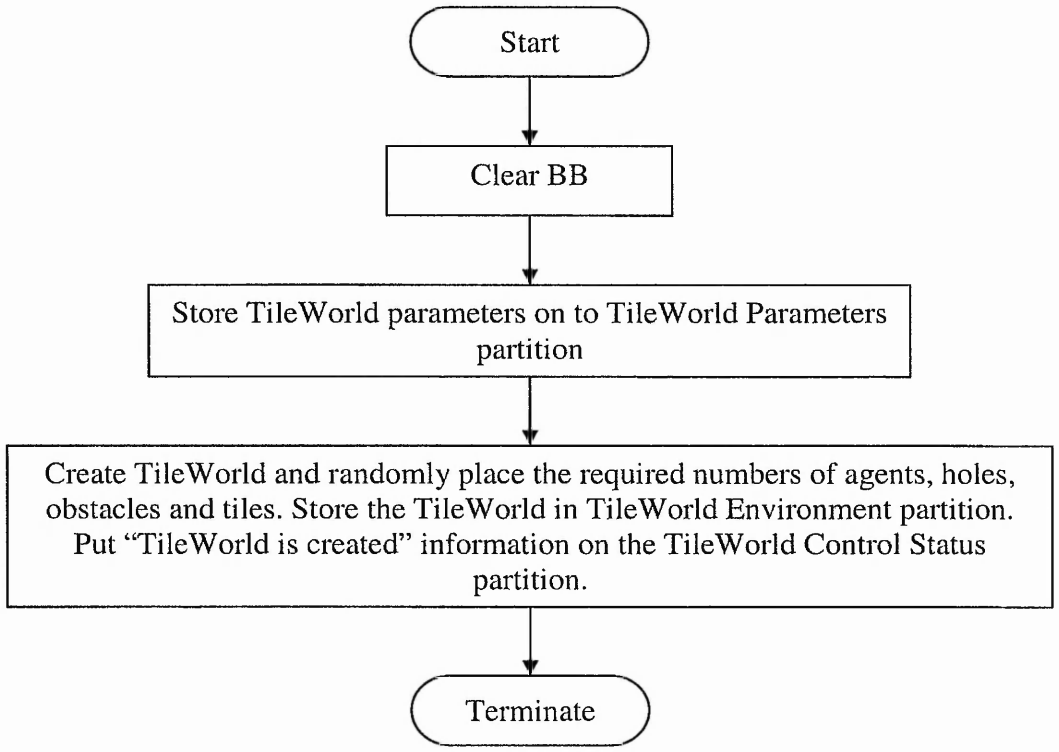


Figure 18. Initiator KS function flow diagram

3.3.2 Display TileWorld KS

The main purpose of the *Display TileWorld KS* is to display the contents of the TileWorld in a graphical form. This was done with the help of the Qt library from Trolltech [95]. Qt is a platform-independent C++ class library that provides easy-to-use graphical function calls for drawing and updating the graphical user interface. The Qt library runs on UNIX, Windows and MacOS operating systems. It is extensively used in

KDE, a Linux graphical desktop environment [96]. As DARBS was written in C++ and implemented in Linux, this makes Qt library an ideal graphical library to use for displaying the TileWorld with its agents, holes, obstacles and tiles.

The *Display TileWorld KS* starts as soon as the TileWorld is created by the *Initiator KS*. The first thing that the *Display TileWorld KS* does is to draw a TileWorld of the size specified in the *TileWorld Parameters* partition. This TileWorld is drawn in a new GUI (Graphical User Interface) window. It then looks for the position of all the agents, holes, obstacles and tiles in the *TileWorld Environment* partition and updates the GUI window accordingly. Finally, it checks and deletes tiles that are on the TileWorld GUI but no longer on the *TileWorld Environment* partition. This is because those tiles must have been picked up by agents. This final check could have also included holes and obstacles if they were dynamic objects. The *Display TileWorld KS* then remains dormant until a change in the *TileWorld Environment* partition has occurred. This is when the BB broadcasts a change in the *TileWorld Environment* partition. The *Display TileWorld KS* can be interrupted whenever the BB broadcasts that a change in *TileWorld Environment* partition has occurred. Figure 19 shows the flow chart of the *Display TileWorld KS*. A detailed list of rules and their description for the *Display TileWorld KS* is in Appendix C.

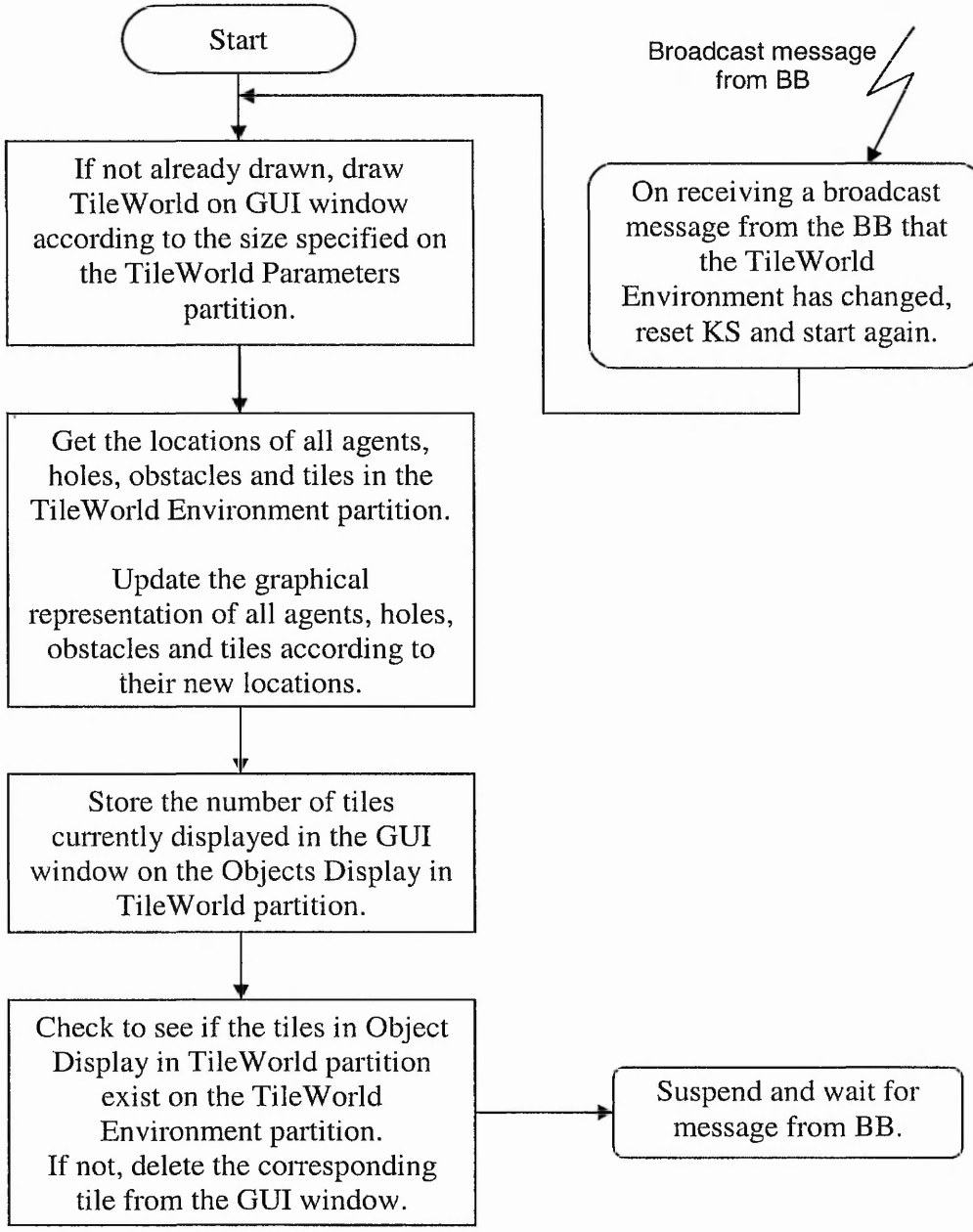


Figure 19. Display TileWorld KS function flow diagram

3.3.3 Agent KS

Each *Agent KS* represents one agent in the TileWorld. As the objectives of every agent in the TileWorld are identical, the rules of each *Agent KS* are identical as well. The only difference being that the rules act on different partitions depending on the agent it is

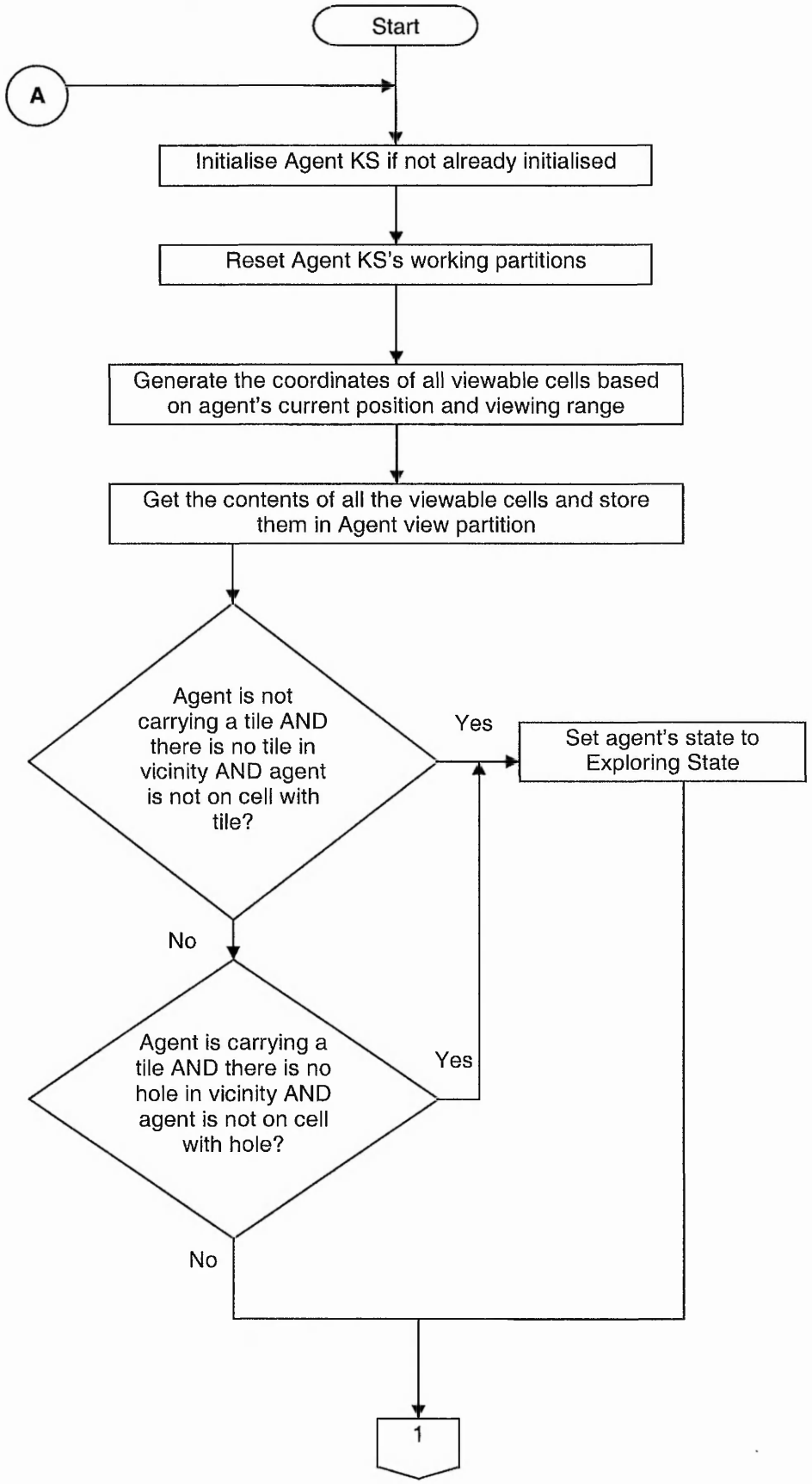
controlling. For example, *Agent1 KS* represents agent1 in the TileWorld and would act on *Agent1 state of mind* partition, *Agent1 view* partition, etc. (see Figure 17).

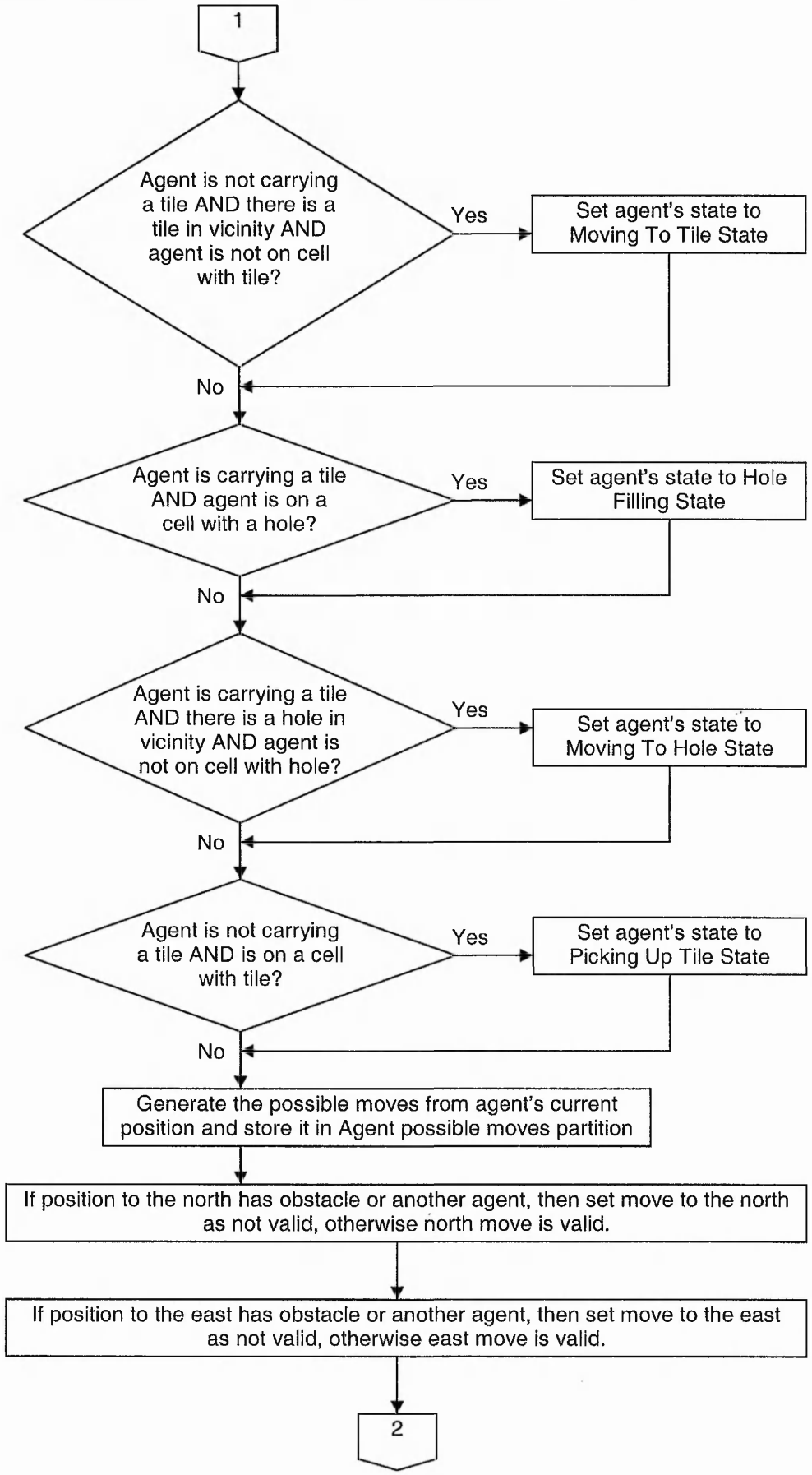
The general algorithm of an *Agent KS* is as follows:

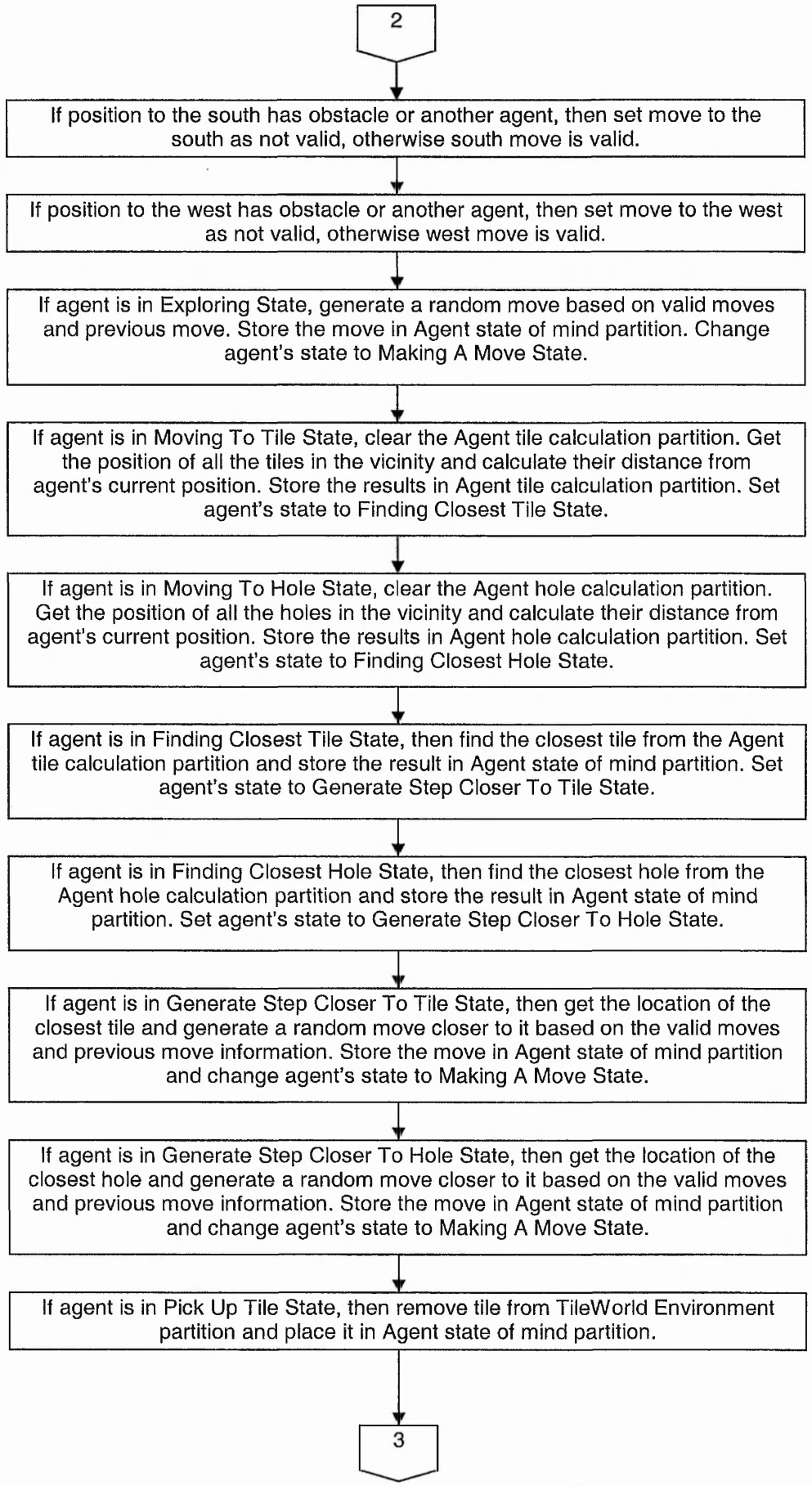
1. Initialise and reset *Agent KS*'s working memory
2. View the surrounding area of agent based on the viewing range limit defined in the agent's working memory.
3. Check to see what state the agent is in based on the following conditions:
    a. If agent is not currently carrying a tile AND there is no tile in vicinity OR if agent is carrying a tile AND there is no hole is in vicinity, then agent is in `Exploring State`.
    b. If agent is not currently carrying a tile AND there is a tile in the vicinity AND there is no tile in the cell that agent is currently occupying, then agent is in `Moving To Tile State`.
    c. If agent is currently carrying a tile AND is occupying a cell with a hole, then agent is in `Hole Filling State`.
    d. If agent is currently carrying a tile AND there is a hole in vicinity AND there is no hole in the cell that agent is currently occupying, then agent is in `Moving To Hole State`.
    e. If agent is not currently carrying a tile AND the cell that it is occupying has a tile, then agent is in `Picking Up Tile State`.
4. If agent is in `Exploring State`, then generate and make a random move.
5. If agent is in `Moving To Tile State`, then make a move towards the closest tile from the current position.

6. If agent is in `Moving To Hole State`, then make a move towards the closest hole from the current position.
7. If agent is in `Picking Up Tile State`, then pick tile up.
8. If agent is in `Hole Filling State`, then drop tile in hole and calculate score.
9. Go back to step 2.

The following flow chart (Figure 20) shows how the general algorithm above is implemented as the *Agent KS*'s rules.


Start
A
Initialise Agent KS if not already initialised
Reset Agent KS's working partitions
Generate the coordinates of all viewable cells based on agent's current position and viewing range
Get the contents of all the viewable cells and store them in Agent view partition
Agent is not carrying a tile AND there is no tile in vicinity AND agent is not on cell with tile?
Yes
Set agent's state to Exploring State
No
Agent is carrying a tile AND there is no hole in vicinity AND agent is not on cell with hole?
Yes
No
1


1
Agent is not carrying a tile AND there is a tile in vicinity AND agent is not on cell with tile?
Yes
Set agent's state to Moving To Tile State
No
Agent is carrying a tile AND agent is on a cell with a hole?
Yes
Set agent's state to Hole Filling State
No
Agent is carrying a tile AND there is a hole in vicinity AND agent is not on cell with hole?
Yes
Set agent's state to Moving To Hole State
No
Agent is not carrying a tile AND is on a cell with tile?
Yes
Set agent's state to Picking Up Tile State
No
Generate the possible moves from agent's current position and store it in Agent possible moves partition
If position to the north has obstacle or another agent, then set move to the north as not valid, otherwise north move is valid.
If position to the east has obstacle or another agent, then set move to the east as not valid, otherwise east move is valid.
2


2
If position to the south has obstacle or another agent, then set move to the south as not valid, otherwise south move is valid.
If position to the west has obstacle or another agent, then set move to the west as not valid, otherwise west move is valid.
If agent is in Exploring State, generate a random move based on valid moves and previous move. Store the move in Agent state of mind partition. Change agent's state to Making A Move State.
If agent is in Moving To Tile State, clear the Agent tile calculation partition. Get the position of all the tiles in the vicinity and calculate their distance from agent's current position. Store the results in Agent tile calculation partition. Set agent's state to Finding Closest Tile State.
If agent is in Moving To Hole State, clear the Agent hole calculation partition. Get the position of all the holes in the vicinity and calculate their distance from agent's current position. Store the results in Agent hole calculation partition. Set agent's state to Finding Closest Hole State.
If agent is in Finding Closest Tile State, then find the closest tile from the Agent tile calculation partition and store the result in Agent state of mind partition. Set agent's state to Generate Step Closer To Tile State.
If agent is in Finding Closest Hole State, then find the closest hole from the Agent hole calculation partition and store the result in Agent state of mind partition. Set agent's state to Generate Step Closer To Hole State.
If agent is in Generate Step Closer To Tile State, then get the location of the closest tile and generate a random move closer to it based on the valid moves and previous move information. Store the move in Agent state of mind partition and change agent's state to Making A Move State.
If agent is in Generate Step Closer To Hole State, then get the location of the closest hole and generate a random move closer to it based on the valid moves and previous move information. Store the move in Agent state of mind partition and change agent's state to Making A Move State.
If agent is in Pick Up Tile State, then remove tile from TileWorld Environment partition and place it in Agent state of mind partition.
3

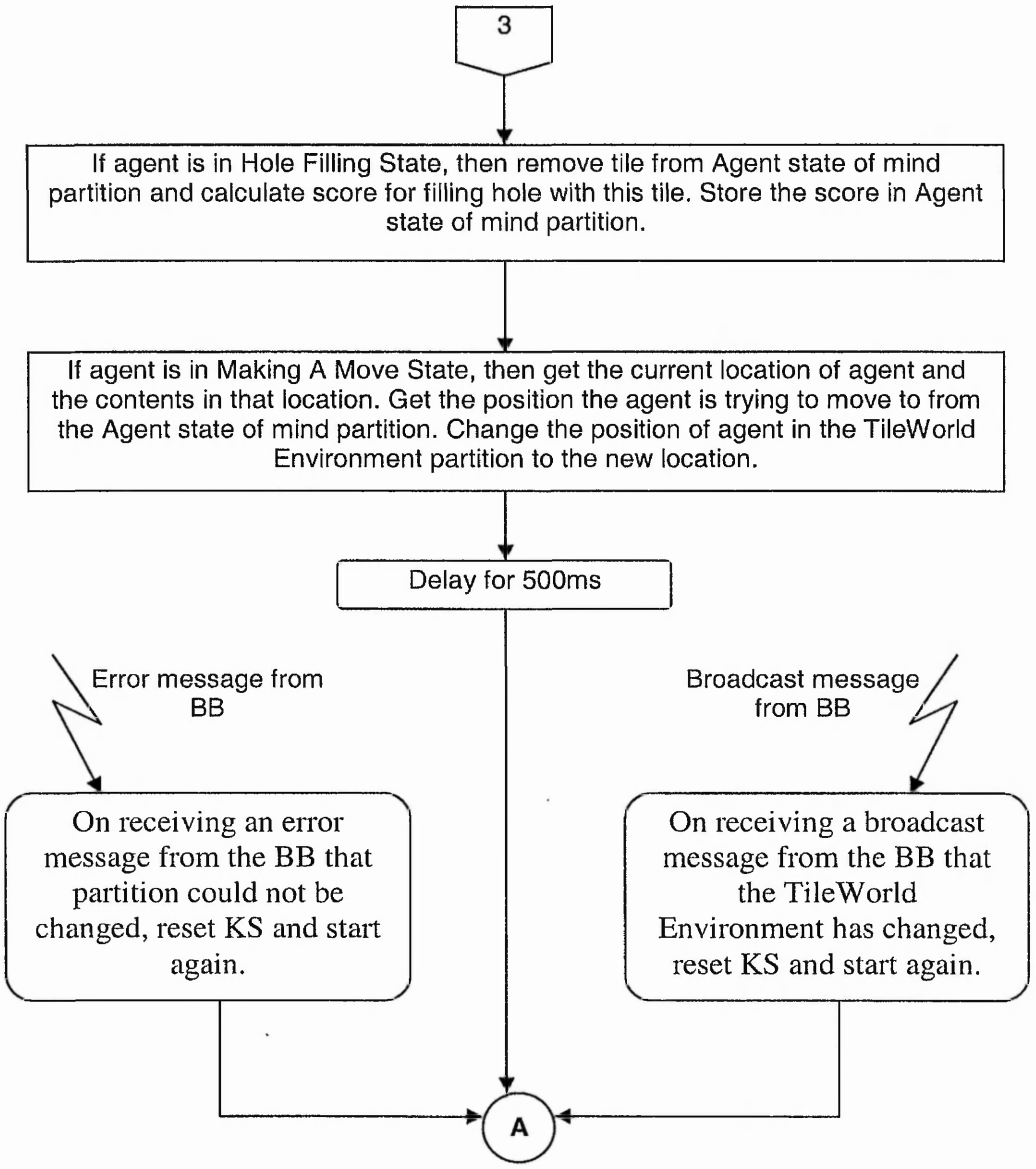


Figure 20. Agent KS function flow diagram

As can be seen in Figure 20, the flow of the functions is sequential and does not branch off. This is because DARBS's current inference engine was designed to test all the rules in sequential order. Therefore, the agent current state was introduced to control the rules that are fired. For example, by setting the agent current state to `Exploring State`, the rule to generate a random move would be fired. If the agent current state was set to `Pick Up Tile State`, then generating a random move rule would not be fired,

instead picking up the tile rule would be fired. A detailed list of rules and their explanation for the *Agent KS* can be found in Appendix C.

The DARBS inference engine allows external functions to be called from within the rule. These functions can be compiled together with the inference engine or can be linked from a dynamic link library. Two examples of external functions called from within the rule are `Generate Random Move` function and `Generate Random Move Closer To Tile/Hole` function. These two functions are important to the *Agent KS* and will be explained in detail in the following sub-sections.

3.3.3.1 Generate random move

The `Generate Random Move` function generates a random move based on the valid moves (i.e. North, East, South and West) and the previous move the agent made. A valid move is into a position that does not have an agent or obstacle in it. The reason the previous move is taken into account is so that the agent would not move back to the cell it just came from unless it is in a dead end. The function first calculates possible moves, i.e. moves that are valid and is not the previous move. If this is the first move, then the agent will not have any previous move, and therefore, would have four possible moves to choose from, assuming that all four moves are valid. In this case, there is a 25% probability of choosing either one of the four possible moves. After this move, all other moves would have zero to three possible moves because of the previous move and the number of obstacles in its path. All possible moves are given equal probability to be chosen except when there are zero possible moves, in which case, the previous move would be chosen.

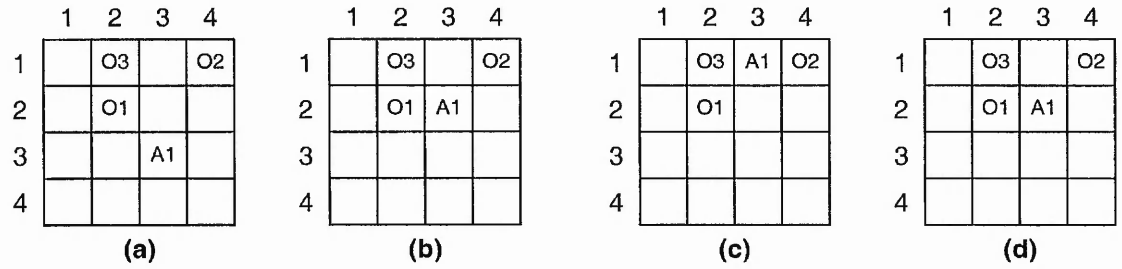


Figure 21. Example of random moves

Figure 21 shows an example of random moves. *Agent1* starts in cell(3,3) and has four possible moves to choose from, i.e. cell(3,2), cell(4,3), cell(3,4) and cell(2,3) (Figure 21 (a)). With a probability of 25%, cell(3,2) was chosen. At cell(3,2), *Agent1* has two possible moves to choose from, i.e. cell(3,1) and cell(4,2) (Figure 21 (b)). With a probability of 50%, cell(3,1) was chosen. At cell(3,1), *Agent1* has zero possible moves to choose from (Figure 21 (c)). Therefore, it chose to take its previous move back to cell(3,2) (Figure 21 (d)).

3.3.3.2 Generate random move closer to tile/hole

The `Generate Random Move Closer To Tile/Hole` function generates a random move closer to a cell location from the current cell location. This move also takes into account the previous move and the valid moves. The first thing this function does is to calculate the distance from the current location to the target cell location. As the TileWorld is made up of grid cells and the agents cannot move diagonally, summing the absolute values of the difference from the current and target location for both the x and y-coordinates would give the distance. This is done using the following equation:

$$dist = |(fromX - toX)| + |(fromY - toY)| \qquad \text{(Equation 1)}$$

where $dist$ is the distance to target cell

$fromX$ is the x-coordinate of the current location

fromY is the y-coordinate of the current location

toX is the x-coordinate of the target location

toY is the y-coordinate of the target location

After calculating the distance, the function then checks for possible moves, i.e. moves that are valid and not the previous move. For each possible move, the distance from the possible move to the target location is calculated. If this new distance is less than the current distance to target, then this possible move is marked as a good move. Because of the grid nature of the TileWorld, there can only be zero, one or two good moves. The random move closer to target is then chosen from the possible moves taking into consideration good moves. The following explains how the move will be selected for each set of possible moves.

- Zero possible moves
    - Select the previous move as the agent is stuck in a dead end. If this is the first move and there is no previous move, then this agent is boxed in by other agents and/or obstacles. In this case, the only thing the agent can do is to remain in the same cell (i.e. new_move = current_location).
- One possible move
    - As there is only one possible move, this move will be selected as the new move to take.
- Two possible moves
    - If there is one good move out of these two possible moves, then the good move will be selected as the new move to take.
    - Else, there are two good moves or no good moves. Either way, each move will have a 50% probability of being chosen as the new move.

- Three possible moves
    - If there is one good move out of these three possible moves, then the good move will be selected as the new move to take.
    - If there are two good moves out of these three possible moves, then each good move will have a 50% probability of being chosen as the new move.
    - Else, there are no good moves. In this case, each possible move will have about 33.33% probability of being chosen as the new move to take.
- Four possible moves
    - If there is one good move out of these four possible moves, then the good move will be selected as the new move to take.
    - If there are two good moves out of these four possible moves, then each good move will have a 50% probability of being chosen as the new move.
    - Else, this is an error as there cannot be no good moves out of four possible moves.

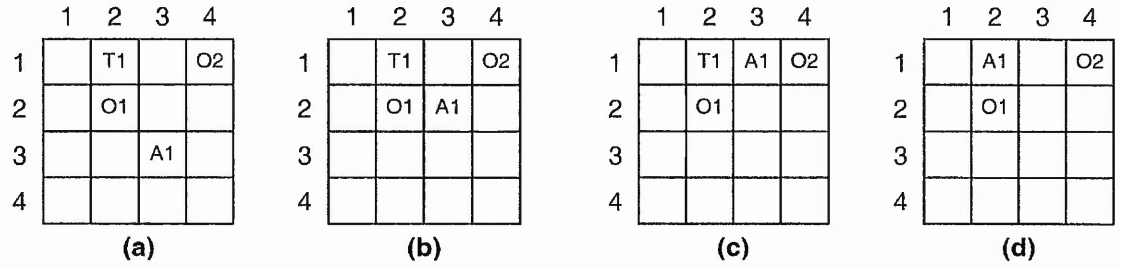


Figure 22. Example of random move closer to tile

Figure 22 shows an example of how random move closer to tile would function. *Agent1* starts from cell(3,3) and finds the closest tile to be Tile1 at cell(2,1) (Figure 22 (a)). *Agent1* calculates the distance to Tile1 to be 3 (i.e. $|(3-2)|+|(3-1)|$). At cell(3,3), *Agent1* has four possible moves and two good moves (cell(3,2) and cell(2,3)). So the two good moves have 50% probability of being chosen as the next move to take. In this

example, cell(3,2) would be chosen as the next move to take (Figure 22 (b)). At cell(3,2), *Agent1* has two possible moves and one good move (i.e. cell(3,1)). Therefore, *Agent1* moves to cell(3,1) (Figure 22 (c)). At cell(3,1), *Agent1* has one possible move of which it is a good move (i.e. cell(2,1)). Therefore, *Agent1* moves to cell(2,1) and picks up Tile1.

3.4 Test and validation

The TileWorld implementation was validated by testing the functionality of each KS on its own and working together. This is known as modular testing and was easily done in DARBS because of its distributed blackboard architecture. The first KS to be tested was the *Initiator KS*. The test for the *Initiator KS* was to make sure that it could set up the TileWorld parameters and generate the TileWorld according to the parameters correctly. To do this test, the BB server was run first followed by the *Initiator KS*. The rules from the *Initiator KS* were all tested and fired by DARBS's inference engine. The *Initiator KS* first cleared the BB and then the parameters of the TileWorld were stored on the *TileWorld Parameters* partition. The TileWorld was then created based on the parameters and the objects in the TileWorld were randomly placed in the TileWorld. The generated TileWorld information was then sent to the BB to be stored in the *TileWorld Environment* partition. The first TileWorld generated was a small 5 × 5 TileWorld. This TileWorld was generated and sent to the BB without any problems. However, when a 20 × 20 TileWorld was generated and sent to the BB, the BB's communication layer could not handle the large message size (i.e. 20 × 20 = 400 location information sent to the BB). Because of this, the *Initiator KS* was modified to send the generated TileWorld information in four parts (i.e. a limit of 100 location information per message). The

Initiator KS could now set up the parameters on the BB, generate the TileWorld accordingly and finally, terminate without any problems.

The *Display TileWorld KS* was the next KS to be tested. The first test for the *Display TileWorld KS* was to make sure that it could display the TileWorld with all its objects in a GUI window correctly. The second test was to make sure that the *Display TileWorld KS* was able to update the GUI window correctly as and when the *TileWorld Environment* partition changes. For the first test, the BB server was run first followed by the *Initiator KS*. When the *Initiator KS* finished setting up the TileWorld and terminated, the *Display TileWorld KS* was then started. The *Display TileWorld KS* correctly displayed the TileWorld and all its objects in a GUI window. For the second test, a terminal client was used to connect to the BB server after the TileWorld has been displayed on the GUI window. Commands were manually sent to the BB via the terminal client to change the location of the agents in the *TileWorld Environment* partition. The *Display TileWorld KS* automatically responded to the change in *TileWorld Environment* partition and updated the GUI window accordingly. The terminal client was then used to send multiple commands to change the locations of the agents in the *TileWorld Environment* partition. It was then observed that although the *Display TileWorld KS* eventually managed to update the GUI window correctly, there was a delay in the update. This delay was due to the broadcasted "`partition TileWorld Environment changed!`" message from the BB that constantly restarted the *Display TileWorld KS*. However, this delay was relatively small and was acceptable as the *Display TileWorld KS* did not have a strict timing requirement. Finally, manual commands were sent to the BB via the terminal client to remove tiles from the *TileWorld Environment* partition. The *Display TileWorld KS* initially had a problem removing the

tiles from the GUI window. This was due to a programming error in the use of the Qt library. This problem was fixed and the *Display TileWorld KS* could now remove tiles from the GUI window correctly.

The last KS to be tested was the *Agent KS*. The first test for the *Agent KS* was to make sure that the *Agent KS* functioned on its own according to the design specification. The BB server and *Initiator KS* were run first and then after the *Initiator KS* had terminated, one *Agent KS* was run. The *Agent KS* was run with debug statements printed out on another text window. This showed the progress of the *Agent KS*. The *Agent KS* functioned according to the design specification, exploring the TileWorld looking for a tile, move towards a tile when it has found one, picking up the tile when it was over one, exploring the TileWorld looking for a hole, moving towards a hole when it has found one, and dropping a tile in a hole when it was over a hole. The whole process was observed on the debug statements. It was also observed that the whole process was relatively slow (i.e. in terms of minutes per cycle) and this was due to the large amount of debug statements.

The second test for the *Agent KS* was the same as the first test except that the *Agent KS* was run together with the *Display TileWorld KS*. The debug statements from the *Agent KS* were reduced to a minimum as the progress of the *Agent KS* could now be displayed graphically by the *Display TileWorld KS*. In this test, the *Agent KS* functioned faster than the first test and according to the design specification. The *Display TileWorld KS* also correctly displayed the progress of the *Agent KS*. The third test was run similar to the second test except that two *Agent KS*s were run together. This test was to test the interaction between *Agent KS*s and to make sure that they function correctly with other

*Agent KS*s. In this test, the size of the TileWorld was reduced to 5 × 5 to increase the interaction between the *Agent KS*s. In this test, the *Agent KS*s functioned according to their design specification. When one *Agent KS* changed the *TileWorld Environment* partition, the other *Agent KS* would restart. This guaranteed that the *Agent KS* worked with as up-to-date information as possible.

For the fourth test, three *Agent KS*s were run together with the *Display TileWorld KS*. During this test, an error occurred that caused the third *Agent KS* to hang half-way. After further investigation, it was discovered that the third *Agent KS* was stuck in a deadlock [97]. This occurred when the main *Agent KS* process was interrupted by a `SIGIO` signal from the kernel while accessing a shared resource. As explained in section 3.2.1.2, when a message is received from the BB, the main process is interrupted and control is passed to the registered signal handler function. The *Agent KS*'s signal handler function accesses the same shared resources as the main *Agent KS* process, therefore, a deadlock occurred when the main process has locked out the shared resource that the signal handler function tried to access. Examples of these shared resources are the heap memory and the standard output device (`STDOUT`). So for example, a deadlock occurs when the main process is interrupted while trying to print out a message on the standard output device and the signal handler function then tries to print out a message on the standard output device. The reason why this deadlock did not occur in the earlier test was because, when there are three *Agent KS*s running together, the number of broadcast messages from the BB increases thus increasing the chances of interrupting the main process while accessing a shared resource.

This problem was solved by changing the way the signal handler function handles signals. The signal handler function now starts a new thread to service the signal and returns control back to the main process to continue its task. By having separate threads running, internal variables that both the signal handler thread and the main process thread access need to be mutually exclusive. This was accomplished by using the POSIX Thread (`pthread.h`) library [98]. This library contains all the necessary thread creating/deleting and mutex locking/unlocking mechanisms. After implementing this new threading signal handler, the fourth test was rerun. This time all three *Agent KS*s and the *Display TileWorld KS* functioned correctly and according to the designed specification. The overall performance of the *Agent KS*s was also observed to have speeded up slightly due to the new signal handler thread.

3.5 Summary

One of the aims of this work was to investigate the performance of a distributed blackboard system in a distributed processing network. To do this, a distributed blackboard system and an application to run on it needed to be chosen. DARBS was chosen as the distributed blackboard system to be used and the TileWorld test-bed was chosen as the application to be run on DARBS [59]. TileWorld test-bed is a well-established multi-agent system test-bed for testing the behaviour of multi-agent systems [87][88]. It is a two dimensional grid world with agents, holes, obstacles, and tiles in it. The objective of the agents is to move around the TileWorld looking for tiles, picking up the tiles and dropping the tiles into holes to score points. The TileWorld was implemented in DARBS by having the agents as KSs and storing the TileWorld environment contents on the BB. The way information was stored on the BB is important as this ultimately affects the performance of the system and intelligibility of

the information on the system. The actual contents of the TileWorld are stored in the *TileWorld Environment* partition on the BB. The information string format for the *TileWorld Environment* partition has been structured in such a way that it enables easy querying of information by the KSs.

There are three types of rule-based KSs that were implemented, they are: *Initiator KS*, *Display TileWorld KS*, and *Agent KS*. The *Initiator KS* sets up the parameters of the TileWorld on the BB and generates the TileWorld based on these parameters. It also randomly places the agents, holes, obstacles, and tiles in the TileWorld. After setting up the TileWorld, the *Initiator KS* terminates. The *Display TileWorld KS* displays the TileWorld and its contents in a GUI (Graphical User Interface) window. This helps users to easily follow the changes in the TileWorld. The *Display TileWorld KS* uses the Qt library from Trolltech for drawing and updating the graphical user interface [95]. The *Display TileWorld KS* starts by drawing the TileWorld according to the parameters set on the BB. It then searches the *TileWorld Environment* partition for the agents, holes, obstacles and tiles and updates the GUI window accordingly. The *Display TileWorld KS* then suspends itself until the BB informs it that the *TileWorld Environment* partition has changed. Whenever the *Display TileWorld KS* receives a "`partition TileWorld Environment changed!`" broadcast message from the BB, it restarts itself and updates the GUI window according to the changes in the *TileWorld Environment* partition.

Each *Agent KS* controls its corresponding agent in the TileWorld. The function and rules of each *Agent KS* are the same. The *Agent KS* starts by examining its surrounding area. If the *Agent KS* is not carrying a tile, then it will look for a tile in the surrounding area. If

there is a tile in the vicinity, it will move towards the tile and pick it up. If there are no tiles in the vicinity, then the agent will randomly explore the surrounding area until it finds a tile. If the agent is already carrying a tile, then it will look for a hole. If the surrounding area has a hole, it will move towards the hole and drop the tile into the hole to score points. If there is no hole in the vicinity, then it will randomly explore the surrounding area until a hole is found. Once the tile has been dropped into the hole and the scores have been calculated, the agent starts looking for another tile again. Whenever the *Agent KS* receives a "`partition TileWorld Environment changed!`" broadcast message from the BB, it will restart itself and check the *TileWorld Environment* partition again. This ensures that all *Agent KS*s are working with the latest *TileWorld Environment* partition information.

After implementing the three KSs, all three KSs were subjected to a functionality test. The *Initiator KS* was the first KS to be tested. A maximum transmitting message size limit was set to overcome the error that occurred when large messages were transmitted to the BB. The second KS to be tested was the *Display TileWorld KS*. Overall, this KS functioned according to the design specification. However, it was noted that there was a slight delay in the graphical update when many changes occur rapidly. This delay was deemed acceptable as there was no strict timing requirement for this KS. The last KS to be tested was the *Agent KS*. Overall, the *Agent KS* functioned according to design specification. However, a deadlock occurred when there were three *Agent KS*s running simultaneously. This was not an error in the *Agent KS*'s design but an error in DARBS's communication model. This deadlock was solved by changing DARBS's inter-process communication (IPC) model to multithreading. With this new IPC model, the test was rerun and the *Agent KS*s all functioned according to the designed specification.

4. Performance of distributed blackboard systems in distributed processing networks

This chapter explains in detail the main experiments that were carried out on a distributed processing network running a distributed blackboard system. The general aim and set-up of these experiments will first be explained followed by a detailed description of the experiments and summary of the results.

4.1 General aim and set-up of experiments

The general aim of the following experiments is to investigate the suitability, potential and characteristics of distributed blackboard systems on distributed processing networks. The results of these experiments would be the basis for determining the suitability of distributed blackboard systems in distributed embedded processing networks. The following general set-up applies to all the experiments carried out in this chapter. The distributed blackboard system used is DARBS running the TileWorld test-bed as explained in chapter 3. From here onwards, TileWorld running on DARBS will be called TileWorld-DARBS. A computer lab with eighteen personal computers (PCs) networked together via an Ethernet 100Mbps switch is used as the distributed processing network for these experiments (see Figure 23).

Figure 23. Computer lab running TileWorld-DARBS

All the PCs used are AMD Athlon 1.67GHz processors with 224 megabytes (MB) of random access memory (RAM) running Red Hat 9 operating system with Linux kernel 2.4. The network layout of the PCs is as shown in Figure 24.

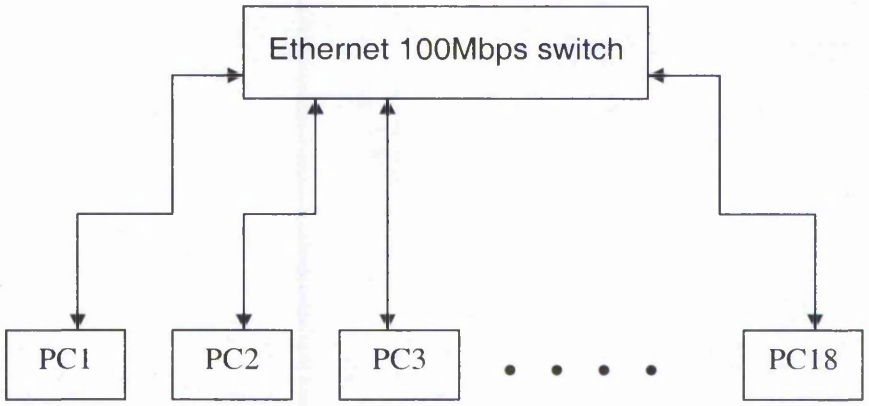


Figure 24. General network layout of experiment PCs

A 20 × 20 TileWorld was created with 40 tiles, 20 holes and 40 obstacles. The position of the tiles, holes, obstacles and the initial positions of the agents were all randomly

generated using the C++ standard random number generator function, `rand()` with a seed of 8. The number of active agents in the TileWorld varied from one to sixteen depending on the experiment. Figure 25 shows the initial layout of the TileWorld used for the experiments.

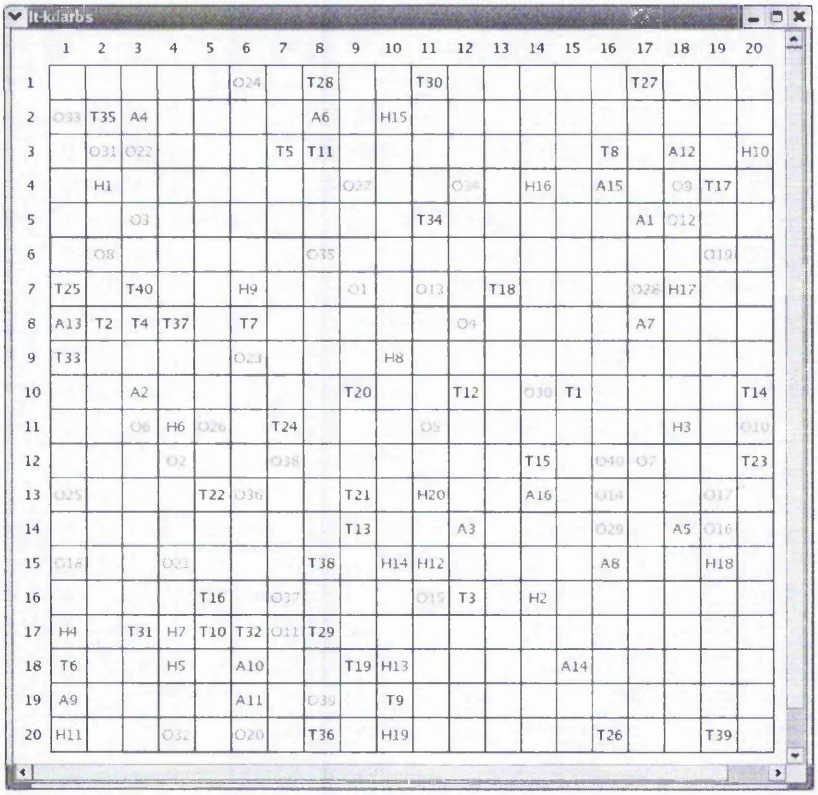


Figure 25. A 20 × 20 TileWorld generated with random seed 8

4.2 Comparing distributed and non-distributed performance

This is the first experiment carried out on TileWorld-DARBS. In this section, the aims and set-up of the experiment will first be explained. Then the results will be discussed and a conclusion reached.

4.2.1 Aims of experiment

The aims of this experiment are to investigate the performance of distributed blackboard systems running in non-distributed set-up and in distributed set-up. To run in non-distributed set-up, TileWorld-DARBS is run on a single processor. To run in distributed set-up, TileWorld-DARBS is run on multi processors (i.e. one processor for the blackboard and one processor for each KS). The distributed set-up will be run in an ideal case, i.e. one processor per process. The parallelism granularity is set to KS level as this will allow the KS programmer to concentrate more on knowledge implementation and ignore fine-grain parallelism issues. Another aim of this experiment is to investigate the effects of adding more KSs to the performance of the distributed blackboard system for both single and multi processors set-up.

4.2.2 Experiment set-up

This experiment is run in two set-ups. In the first set-up, TileWorld-DARBS is run with one to sixteen agents on a single processor. In the second set-up, TileWorld-DARBS is run with one to sixteen agents on multi processors (i.e. a new processor is added to the network for every new agent KS). Figure 26 illustrates the single processor set-up with different number of agents.

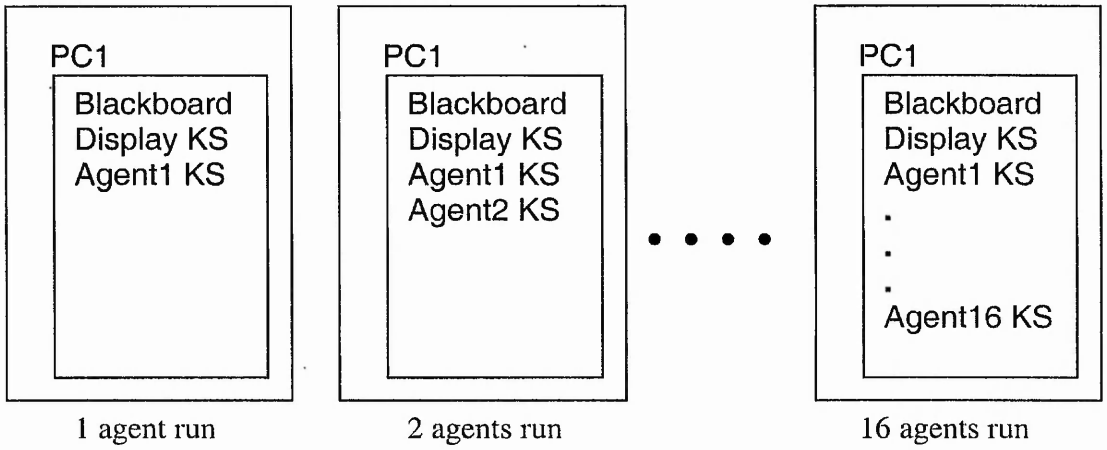
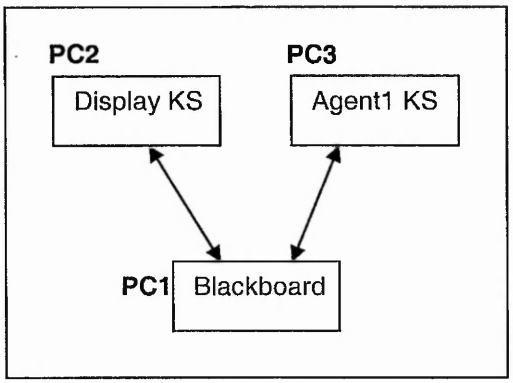
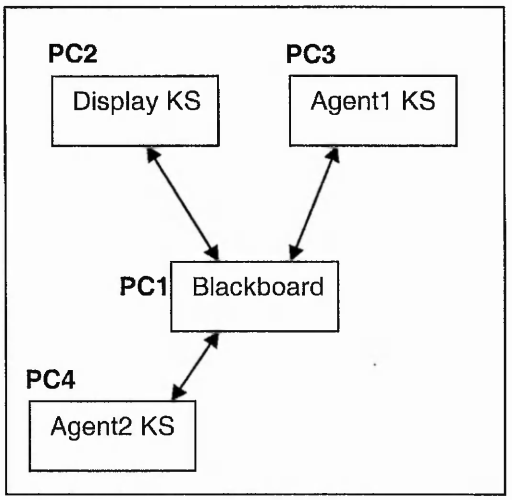


Figure 26. Single processor set-up for different number of agents

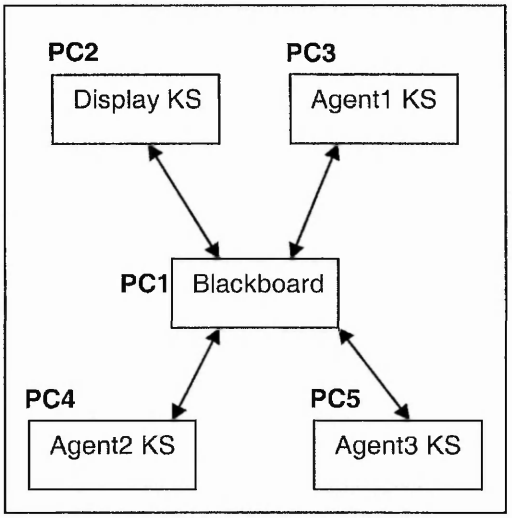
Figure 27 shows the multi processors set-up with different number of agents. For every run of the experiment, each agent's average time per move is calculated over 50 moves. The overall mean time per move for each run is then calculated as the average of all the agents' mean times per move. The time per move is calculated by subtracting the time of an agent's move from the time of its subsequent move. An agent is considered to have made a move when it has changed the TileWorld environment (i.e. moved to another cell, picked up a tile, or dropped a tile into a hole). Restarts due to other agents changing the TileWorld are not considered as moves.



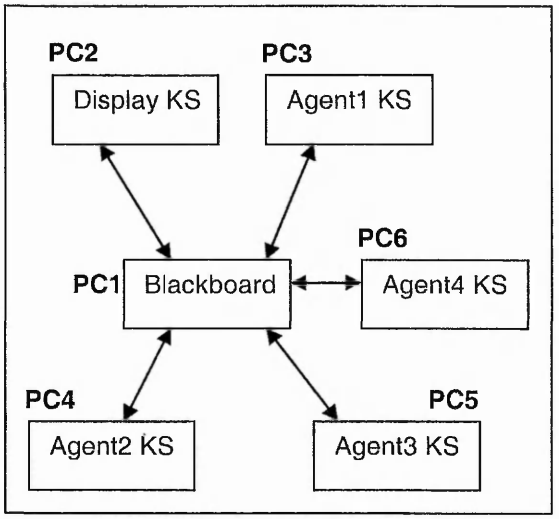
1 agent run



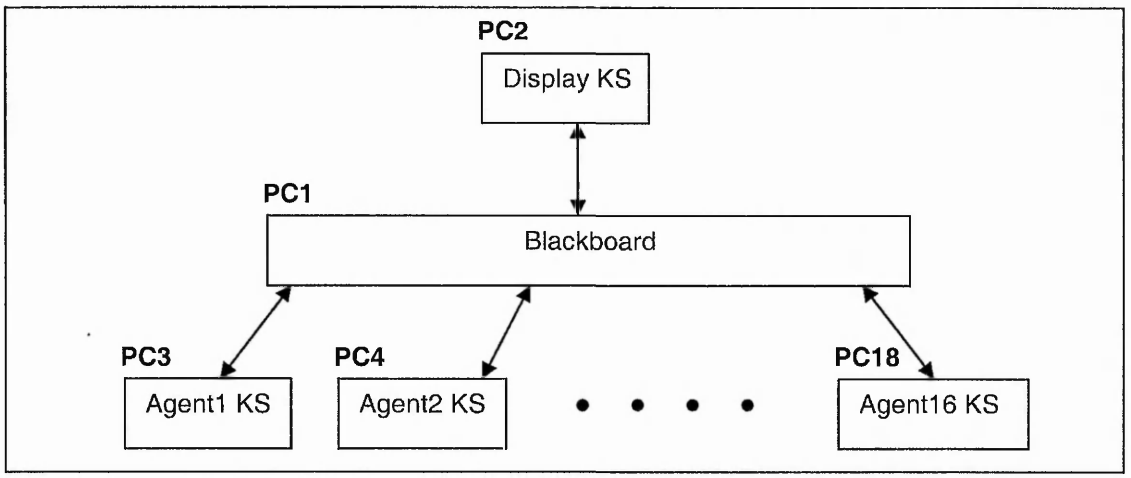
2 agents run



3 agents run



4 agents run



16 agents run

Figure 27. Multi processors set-up for different number of agents

4.2.3 Results and discussion

Figure 28 shows the results of the single processor set-up of the experiment. A polynomial function is fitted onto the results to show the general trend of the results. The polynomial function is used instead of a linear function because the coefficient of determination (R^2) [99] for the linear function is 0.9806 (see Appendix D) whereas the coefficient of determination (R^2) for the polynomial function is 0.9981. The error bars show the standard error of mean [100] for each run.

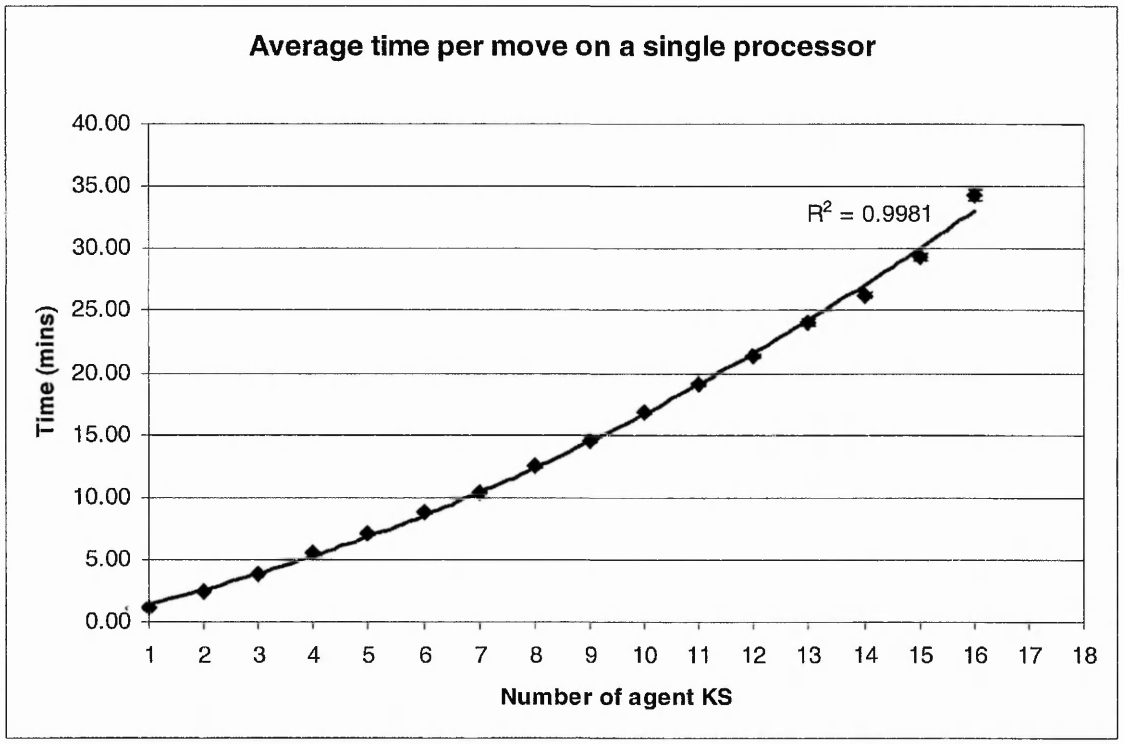


Figure 28. Average time per move for different number of agent KSs on a single processor

From Figure 28, it can be seen that the average time per move increases slightly more than linearly as the number of agent KSs increases. This is as expected, as the number of agent KSs increases, the single processor needs to time-slice [101] between more processes. For each time-slice, the processor needs to save the current process's context

before switching to the next process's context. This is called context-switching [102] and this takes up processing time. A detailed table of the results can be seen in Appendix E.

The standard error of mean [100] is calculated as:

$$\sigma_M = \frac{\sigma}{\sqrt{N}} \qquad \text{(Equation 2)}$$

where σ_M is the standard error of mean

σ is the standard deviation

N is the sample size (in this case it is 50)

The standard error of mean also increases as the number of agent KSs increases although this is not clear in Figure 28 due to the scale of the graph. The exact values can be seen in the table of results in Appendix E. The increase in standard error of mean is due to the increase in the standard deviation. This increase was observed to be because of the interaction between the agent KSs. As the number of agent KSs increases, the agent KSs start to compete among themselves. Take Figure 29 for example, where Agent1 and Agent2 are competing to get Tile1. Agent1 takes a step closer to Tile1 by moving to cell(2,2). Agent2 has two possible moves (cell(2,2) or cell(3,1)) that will bring it closer to Tile1 and has decided randomly to move to cell(2,2) only to discover that Agent1 is already in that cell (Figure 29 (b)). Agent2 now has to restart and think again where to move. Agent2 then decides to move to cell(3,1) (Figure 29 (c)). Agent1 now moves to cell(3,2) to pick up Tile1. Agent2 tries to move to cell(3,2) to pick up Tile1 but discovers Agent1 is already in that cell. Agent2 now has to restart and think again. As restarts do not count as moves, Agent2 has taken a long time to make its move (i.e. from cell(2,1) to cell(3,1)). This competition only occurs occasionally when the agent KSs are close to

each other. Most of the time they are apart as the TileWorld is relatively large compared to the number of agents in the TileWorld.

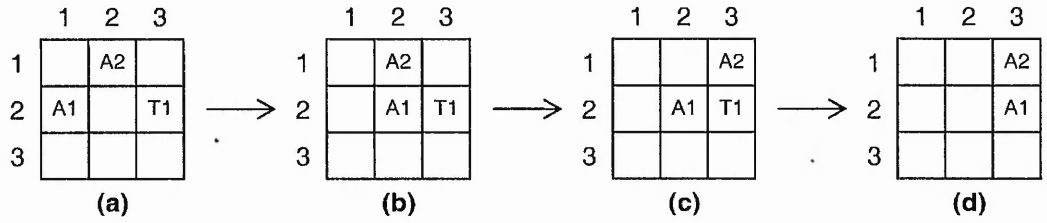


Figure 29. Example of competing agent KSs

Figure 30 shows the results of the multi processors set-up of the experiment. A linear function is fitted onto the results to simplify and show the general trend of the results. This is because the polynomial function trend line produces an equation and R^2 value (see Appendix D) that is very similar to the linear function. The error bars show the standard error of mean for each run. The results on multi processors show a more linear trend compared to the single processor. This is because on multi processors there is no time-slicing between the processes. The increase in average time per move is mainly due to the communication time between the processors (a more detailed explanation can be found in section 4.3.3). The standard error of mean also increases as the number of agent KSs increases and this is due to the same reason as that on the single processor. A detailed table of the results can be seen in Appendix E.

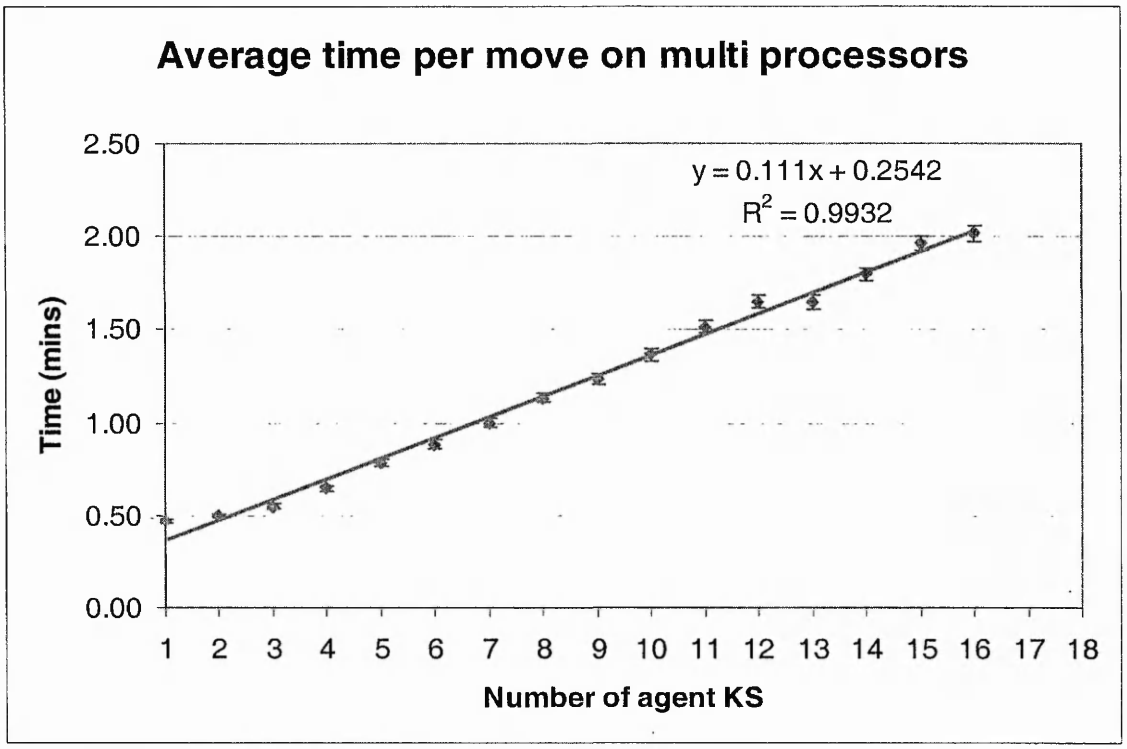


Figure 30. Average time per move for different number of agent KSs on multi processors

To illustrate the effects of adding KSs further, the average time per move for each run is normalised to the average time per move for one agent KS for both the single and multi processors experiments. This will show the affects of adding new agent KSs to the system. The normalised value is calculated as:

$$t_N = \frac{NAgentTime}{OneAgentTime} \qquad \text{(Equation 3)}$$

where $NAgentTime$ is the mean time per move for N agent KSs

$OneAgentTime$ is the mean time per move for 1 agent KS

t_N is the normalised time

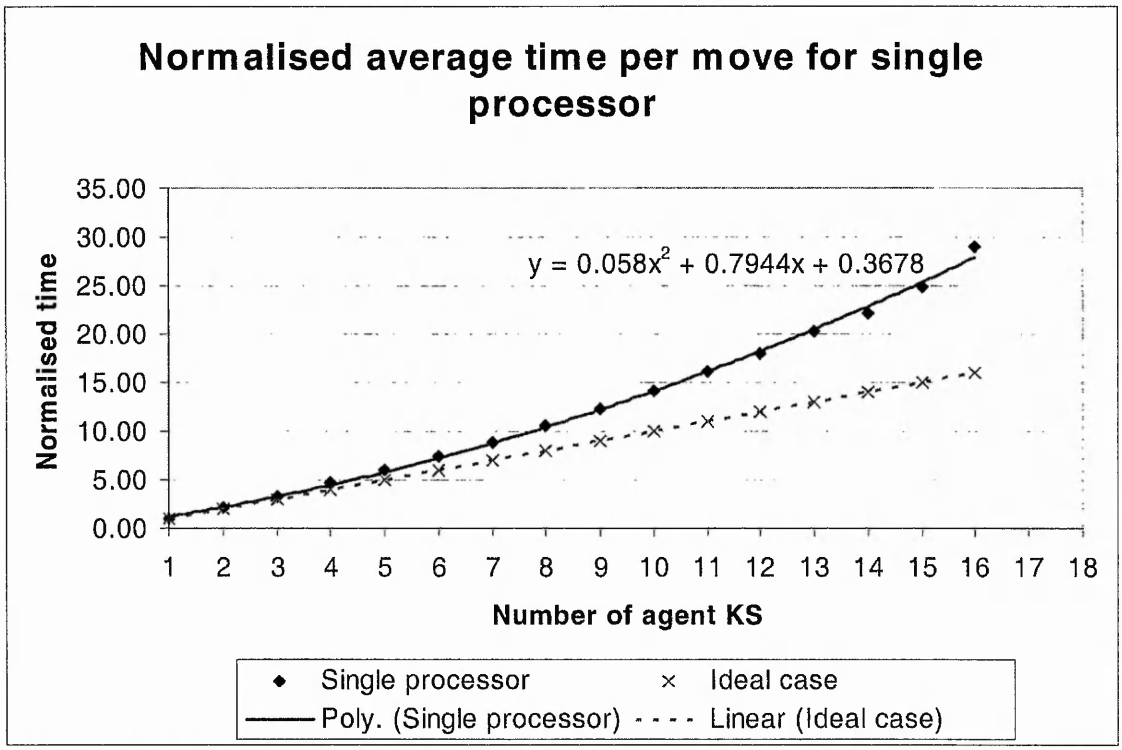


Figure 31. Normalised time for single processor

Figure 31 shows the normalised time with increasing number of agent KSs on a single processor. A second order polynomial function is fitted onto the normalised values. The effects of adding more agent KSs on a single processor can clearly be seen in Figure 31. The slow-down as the number of agent KSs increases is approximately linear until four agent KSs. By adding four agent KSs in the TileWorld, the average time per move is approximately five times slower than one agent KS in the TileWorld. This increase in time is not linear and at sixteen agent KSs, the slow-down is close to 30. The gradient of the polynomial function can be calculated by differentiating it as follows:

$$y = 0.058x^2 + 0.7944x + 0.3678$$

$$\frac{dy}{dx} = 0.116x + 0.7944$$

The derivative here shows roughly the rate of slow-down as the number of agent KSs increases up to sixteen agent KSs. In the ideal case this slow-down should be linear as

the single processor would spend equal amount of processing time for every agent KS in the TileWorld. However, because of the large context-switching overhead and an inefficient scheduling algorithm of the kernel, this slow-down is a polynomial function. The Linux kernel's scheduling algorithm dynamically recalculates each process's priority every epoch and as the number of processes increases, this recalculation becomes a burden [103]. A detailed table of results can be seen in Appendix E.

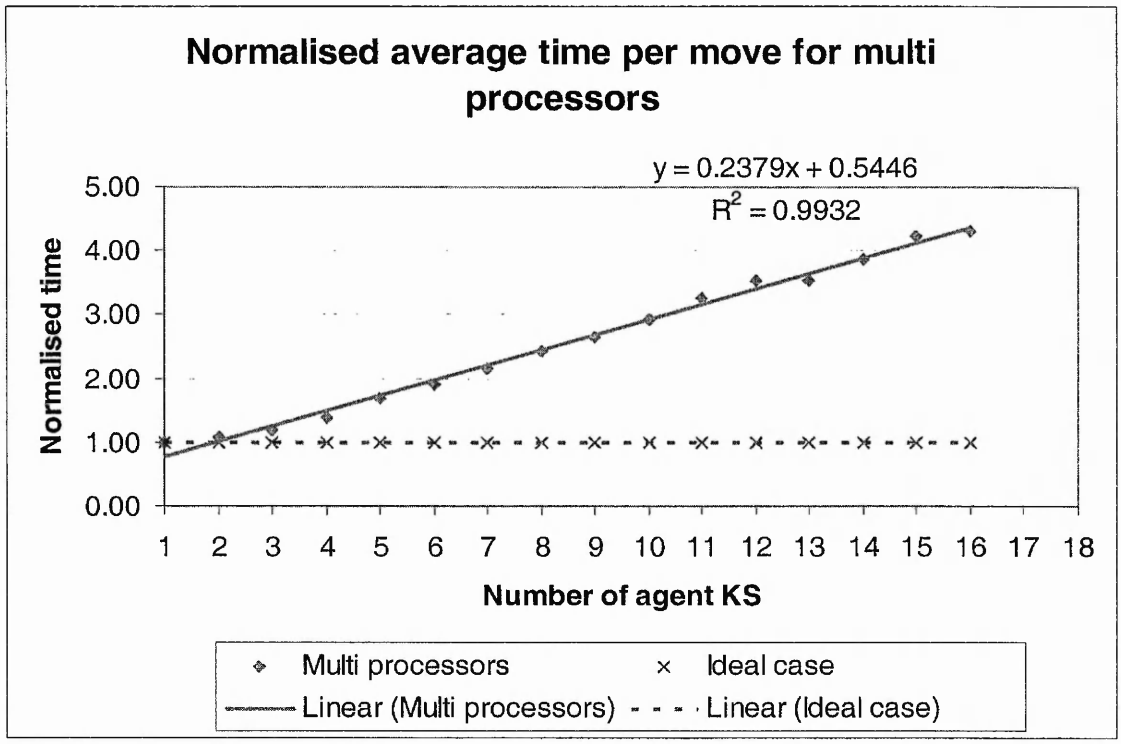


Figure 32. Normalised time for multi processors

Figure 32 shows the normalised time for the multi processors set-up with a linear function fitted onto the normalised values. The reason why a linear function is used is the same as the reason given for Figure 30. An ideal case is also plotted as a straight line with a normalised time value of 1. In the ideal case, there should not be any slow-down as a new processor is assigned to every new agent KS in the TileWorld. From the experiment, the first two values are relatively close to the ideal case. However, from

three agent KSs onwards, the slow-down is approximately linear with a gradient of about 0.238. This slow-down is mainly due to the communication overhead and the serial access to the blackboard. The communication overhead is mainly caused by the extra messages that the blackboard needs to send out during a broadcast message. This is because DARBS accomplishes a broadcast message by sending the same message to all the KSs in turn. The blackboard serial access slow-down, on the other hand, is caused by the increase in queuing time needed to access the blackboard that results from the increase in the number of agent KSs. Compared with a single processor with up to sixteen agent KSs, the rate of slow-down on multi processors is a lot less. This is shown by the gradient of the two trend lines ($\frac{dy}{dx} = 0.116x + 0.7944$ and 0.238 respectively). At sixteen agent KSs, the difference between the measured time and the ideal time is;

For multi processors:

$$4.3214 - 1.0000 = \underline{3.3214}$$

For single processor:

$$28.9718 - 16.0000 = \underline{12.9718}$$

This shows that at sixteen agent KSs, the results on the single processor is a lot further from the ideal case than that of the multi processors. A detailed table of results can be seen in Appendix E.

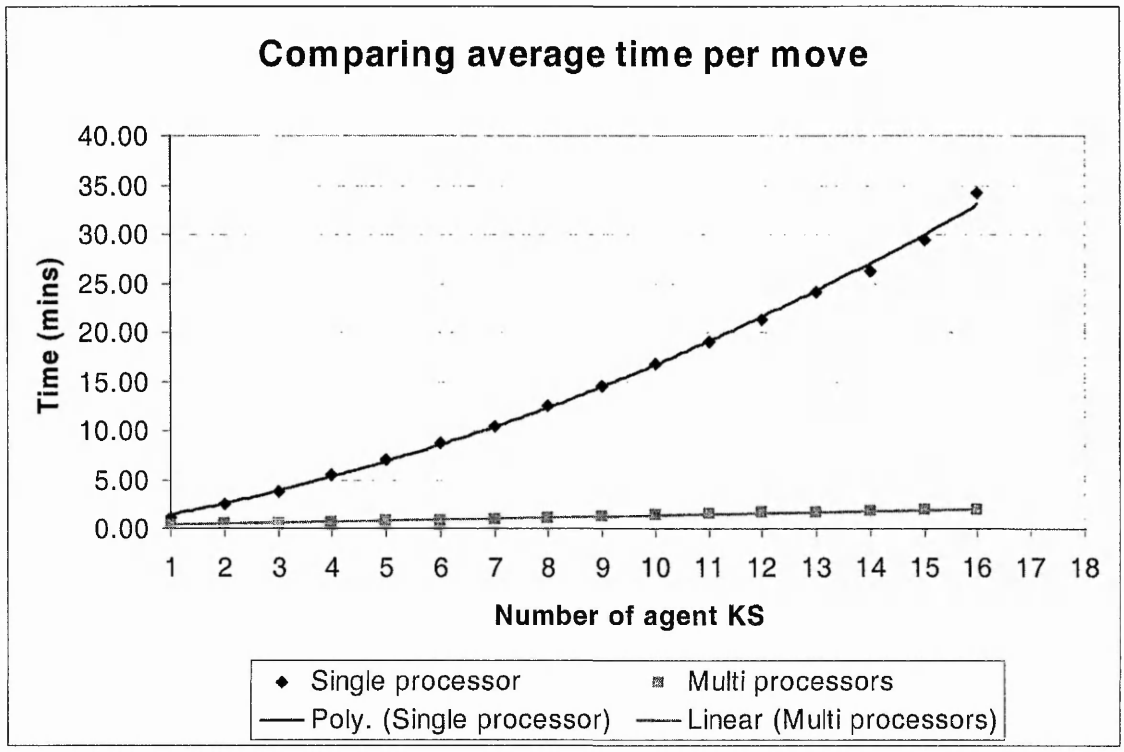


Figure 33. Single and multi processors average time per move

Figure 33 compares the average time per move on a single and multi processors. A polynomial function and a linear function are fitted to the results of the single processor and multi processors respectively. It can clearly be seen that as the number of agent KSs increases, the difference in time between single and multi processors increases. For example, at one agent KS, the difference is:

$$1.183 - 0.467 = \underline{0.716 \text{ minutes}};$$

and at sixteen agent KSs, the difference is:

$$34.283 - 2.017 = \underline{32.266 \text{ minutes}}$$

This shows that the overhead from context-switching on the single processor is more than the communication and serial access overhead on the multi processors as the number of agent KSs increases. For one agent KS, the ratio between single and multi processors is:

$$\frac{1.183}{0.467} = \underline{2.533}$$

and for sixteen agent KSs, the ratio is:

$$\frac{34.283}{2.017} = \underline{16.997}$$

This means that with one agent KS, the multi processors set-up is about two and a half times faster than the single processor set-up and this continues to increase up to approximately seventeen times faster with sixteen agent KSs (as can be seen in Figure 33). A detailed table of results can be seen in Appendix E.

4.2.4 Conclusion

From this experiment, it is evident that the performance of the distributed blackboard system in a distributed set-up is better than on a non-distributed set-up. The performance of both distributed and non-distributed set-ups are far from their ideal case. For the distributed set-up, this is because of the communication and serial access overheads. For the non-distributed set-up, this is because of the context switching overhead and the kernel's inefficient scheduling algorithm. Although the scheduling algorithm can be improved, the performance of the non-distributed set-up would still be far from the ideal case as context switching between processes takes up relatively large amount of processing time. Comparing the difference between the actual performance results and the ideal case, the distributed set-up has a smaller difference compared to the non-distributed set-up. This is true for up to sixteen agent KSs. However, there will be a

point when the blackboard saturates. This will be when the number of requests from KSs is more than the blackboard can handle. The performance at and beyond this point will be difficult to predict without further experiments (see future work section 6.3), however, the third experiment in section 4.4.3 identifies the saturation point for this particular system set-up.

The standard deviation of the results increases as the number of agent KSs increases. This was observed to be due to the increase in number of restarts. The restarts were caused by the competitive interaction between agent KSs. This means that as the number of agent KSs increases, the variation in the readings increases. The restart algorithm can be optimised but this will only speed up the restart algorithm. A better way to overcome this is to change the behaviour of the agent KSs to cooperative but this will be the subject of agent behaviour research. Currently, agent behaviour is the subject of numerous researches [104][105][106][107] and this is beyond the scope of this thesis.

The effects of adding agent KSs to the TileWorld in the non-distributed set-up is a polynomial slow-down function. In the distributed set-up on the other hand, the slow-down is approximately a linear function. The rate of slow-down of up to sixteen agent KSs for the non-distributed set-up is approximately $\frac{dy}{dx} = 0.116x + 0.7944$ and for the distributed set-up, it is approximately 0.238. This means that by adding more agent KSs to the non-distributed set-up, the slow-down increases linearly but for the distributed set-up, the slow-down remains constant at 0.238. This shows that for a large number of agent KSs, it is better to run in the distributed set-up. However, the down side of the distributed set-up is that there is an assumption of unlimited processor resource. In practical application cases, there is a limited processor resource. Therefore, the KS

processes need to be shared among the available processor resource. The next sets of experiments investigate the speedup and efficiency of sharing the KS processes among varying number of processors.

4.3 Speedup and efficiency of varying number of agent processors

This is the second experiment carried out on TileWorld-DARBS. In this section, the aims and set-up of the experiment will first be explained. Then the results will be discussed and finally a conclusion of the results will be presented.

4.3.1 Aims of experiment

The aims of this experiment are to investigate the speedup factor and efficiency of varying number of agent processors. An agent processor (AP) is a processor dedicated to run one or more agent KSs. The reason for not considering the Display KS is because it is only there to display the contents of the TileWorld; without it the TileWorld would still function. In general, speedup is a measure of the speed improvement of parallel processing over sequential processing [108] and it is calculated as:

$$Speedup = \frac{time_s}{time_n} \qquad \text{(Equation 4)}$$

where $time_s$ is the execution time on a single processor

$time_n$ is the execution time on N processors

$Speedup$ is the speedup factor

There are different sub-categories of speedup and relative speedup [108] is used here as the interest is in the speed increase using the same distributed blackboard system. Relative speedup is defined as the speed improvement of running a parallel algorithm in

parallel over running the same parallel algorithm in sequential [71]. For simplicity, relative speedup will just be referred to as speedup from here onwards. Efficiency is the measure of the amount of speedup gain per processor used [109] and it is calculated as:

$$Efficiency = \frac{Speedup}{N_{processor}} \qquad \text{(Equation 5)}$$

where $Speedup$ is the speedup factor

$N_{processor}$ is the number of processors used

$Efficiency$ is the efficiency

The optimal distribution of KSs and its scalability will also be investigated using these two performance metrics. The speedup obtained will also be compared with the suggested speedup in [11].

4.3.2 Experiment set-up

This experiment is run in seven different set-ups. For the first set-up, ten agent KSs are run in one to ten numbers of agent processors (APs). The average time per move is calculated for each run. The speedups and efficiencies are then calculated from the average time per move. For the second set-up, eleven agent KSs are run in one to eleven numbers of APs and their speedups and efficiencies are calculated. This is repeated for all the other set-ups until sixteen agent KSs in one to sixteen numbers of APs. As the parallelism granularity is set at KS process level, the maximum number of APs that can be used is the number of agent KSs being run. This can be clearly seen in Figure 34.

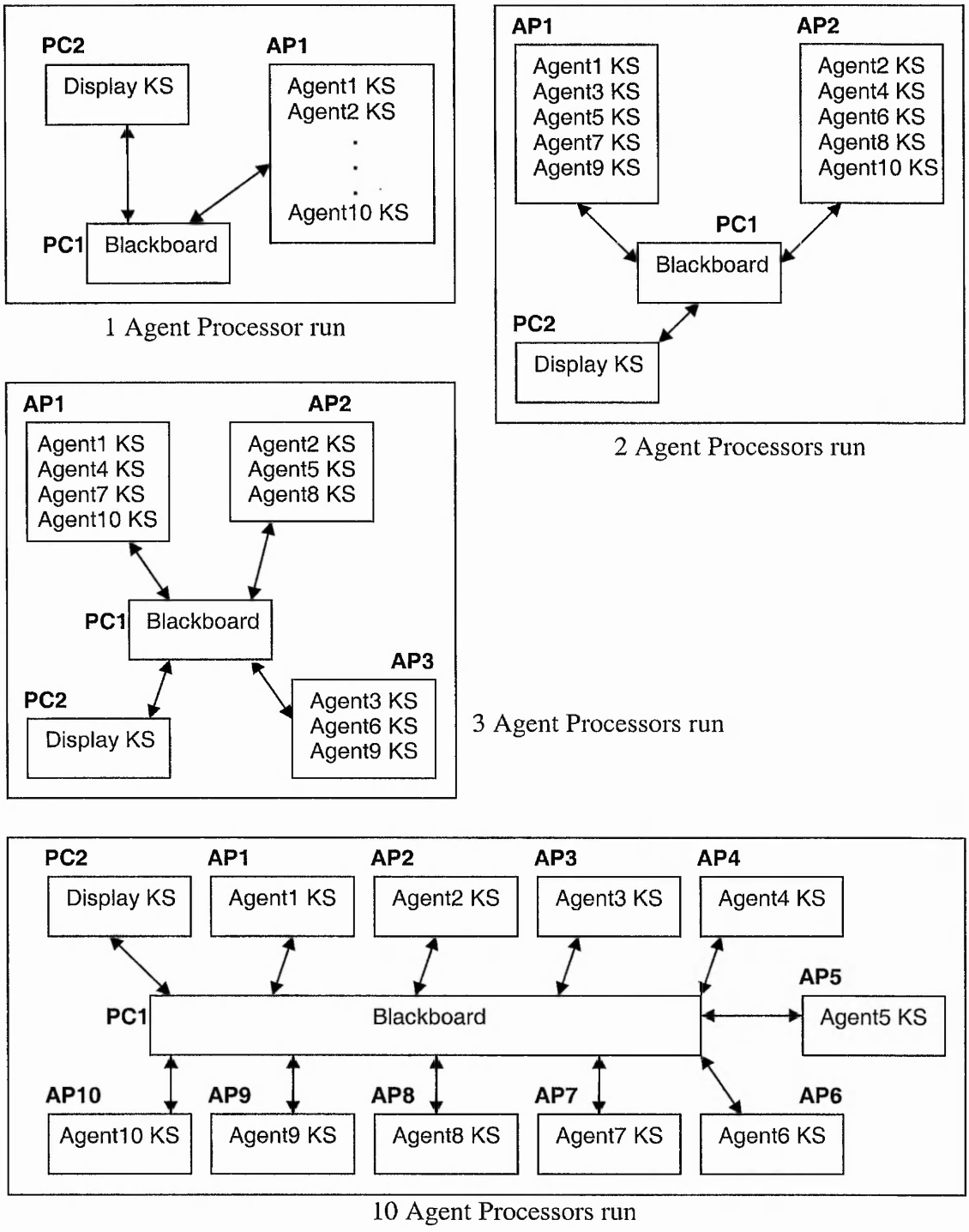


Figure 34. Ten agent KSs set-up with diffirent numbers of APs

Figure 34 shows how ten agent KSs set-up is run in one to sixteen APs. As can be seen also, the available agent KSs are distributed as evenly as possible across the available APs.

4.3.3 Results and discussion

Figure 35 to Figure 41 show the results of the ten to sixteen agent KSs set-ups in two y-axis charts respectively. For each chart, a polynomial function is fitted on to the speedup results to show the trend of the results (dotted line). Another polynomial function is fitted on to the efficiency results to show the trend of the efficiency results (solid line). The speedup values for the dotted line are shown on the left y-axis and the efficiency values for the solid line are on the right y-axis.

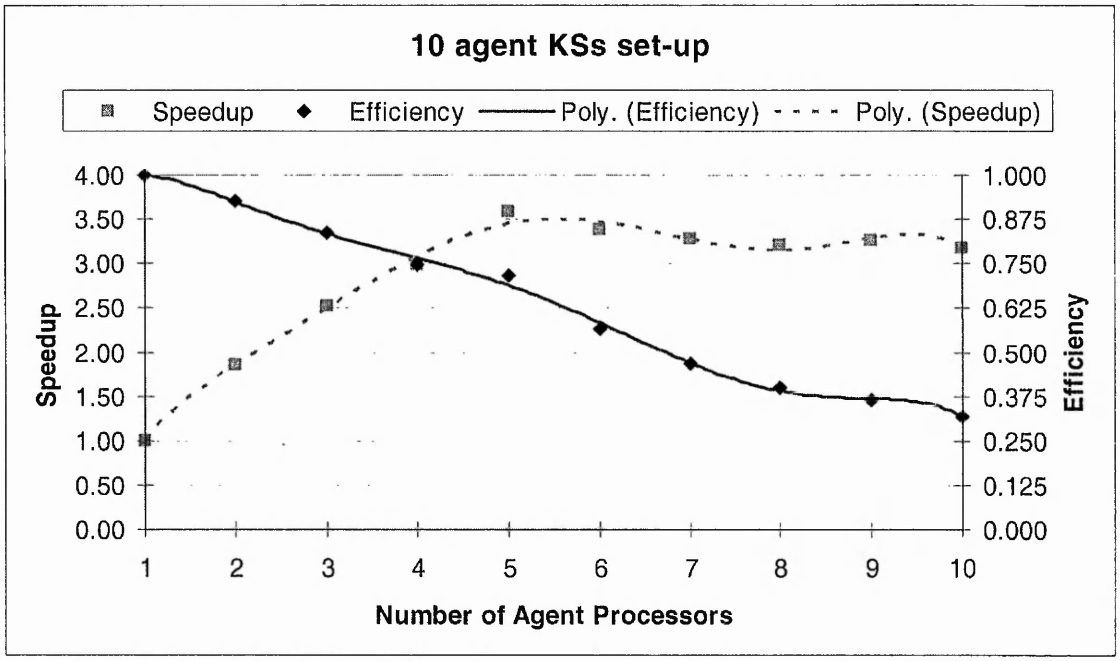


Figure 35. Speedup and efficiency for 10 agent KSs set-up

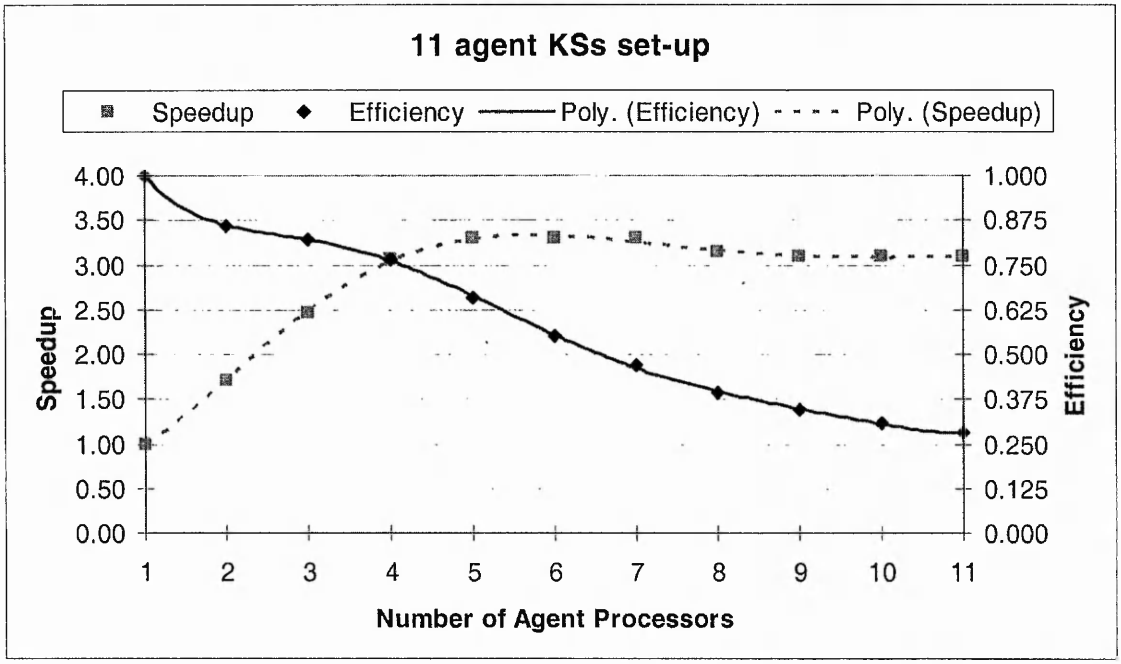


Figure 36. Speedup and efficiency for 11 agent KSs set-up

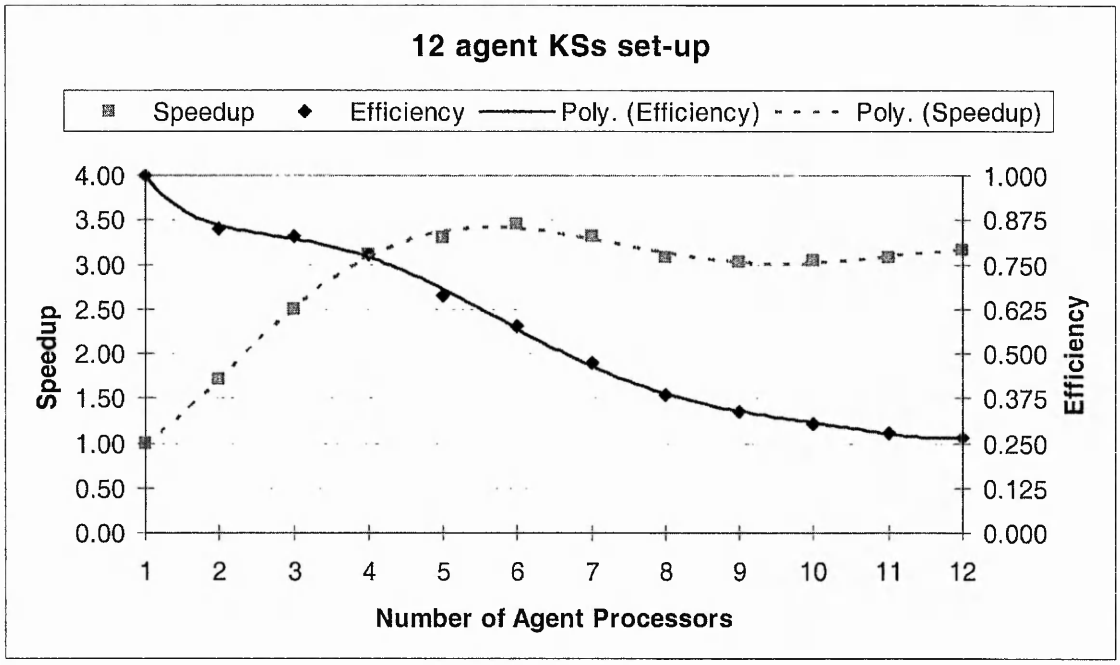


Figure 37. Speedup and efficiency for 12 agent KSs set-up

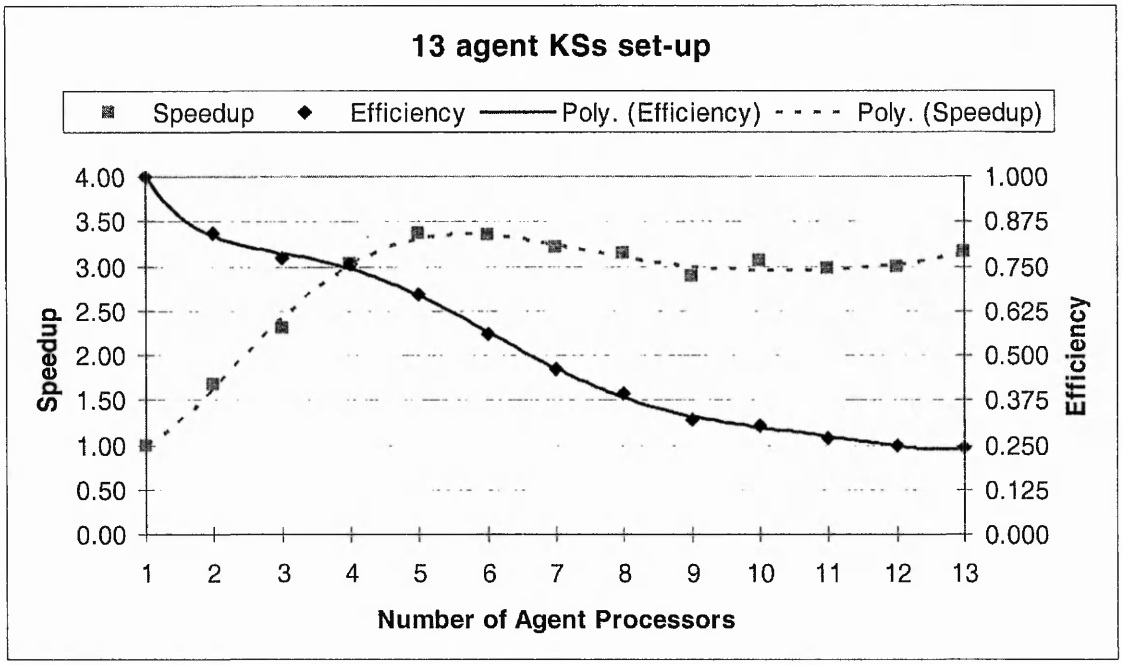


Figure 38. Speedup and efficiency for 13 agent KSs set-up

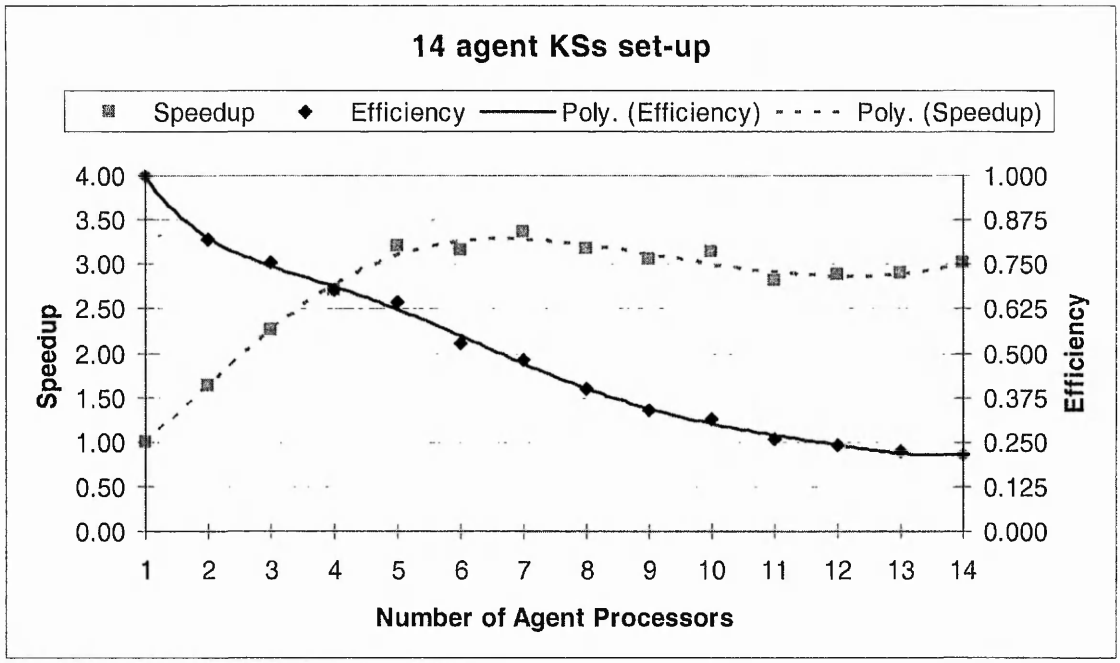


Figure 39. Speedup and efficiency for 14 agent KSs set-up

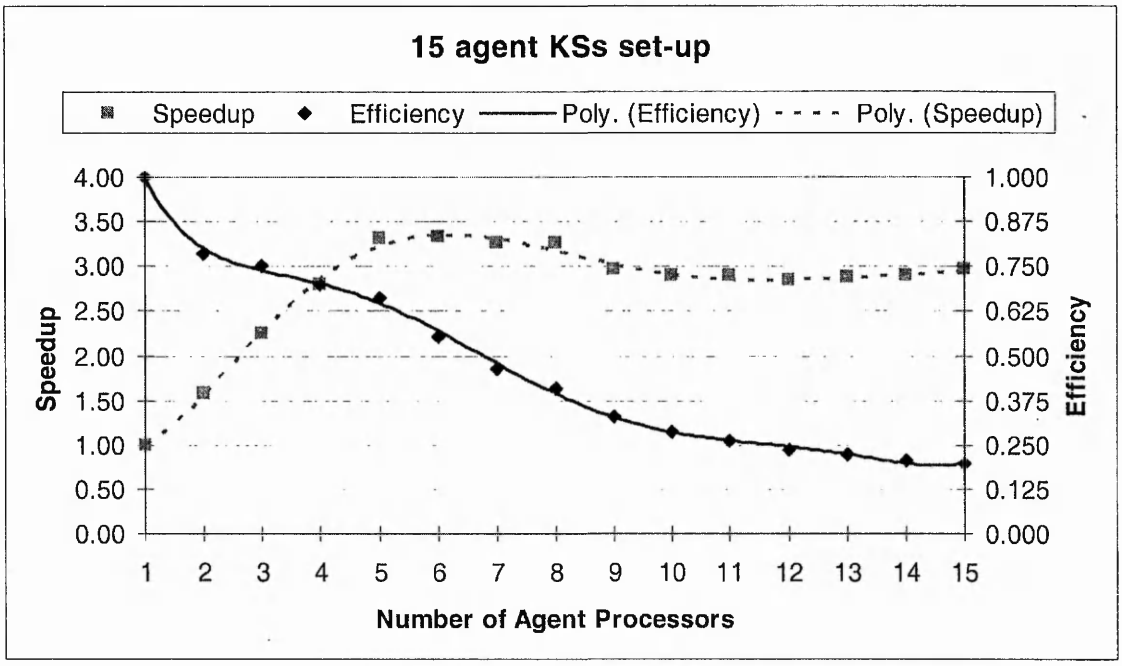


Figure 40. Speedup and efficiency for 15 agent KSs set-up

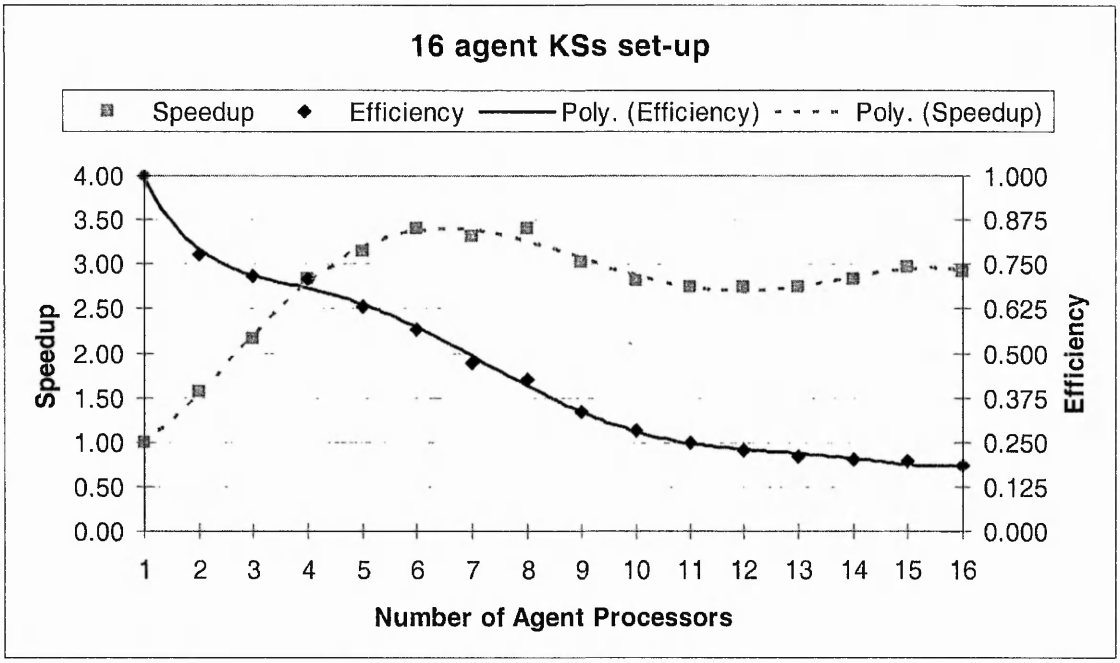


Figure 41. Speedup and efficiency for 16 agent KSs set-up

The results for all the different agent KSs set-ups show that the speedup increases as the number of APs increases up to a peak after which the speedup slowly levels off. For ten

agent KSs set-up (Figure 35), the peak is at five APs with a value of about 3.59 after which it levels off to about 3.18. For eleven and twelve agent KSs set-ups (Figure 36 and Figure 37), the peaks are at six APs with a value of about 3.30 and 3.45 respectively. After the peak, both speedups level off to about 3.10 (eleven agent KSs set-up) and 3.17 (twelve agent KSs set-up). However, for thirteen and fourteen agent KSs set-ups (Figure 38 and Figure 39), the peak is at five and seven APs respectively with a value of about 3.36. Both set-ups then level off to about 3.17 and 3.03 respectively. Finally for fifteen and sixteen agent KSs set-ups (Figure 40 and Figure 41), the peaks are at six and eight APs with a value of about 3.32 and 3.40 respectively. After the peak, both speedups level off to about 2.96 and 2.92 respectively.

The small oscillated results after the peaks as the speedups level off are due to the high standard deviation as explained in section 4.2.3. The initial increase in speedup before the peak is because of the reduced number of KS processes that each AP needs to process. However, after the peak (maximum point), further increase of APs reduces and levels off the speedup. This is because as more APs are connected to the BB, the query requests to the BB become more frequent. The BB is interrupted by each query request to put that query request on to a queue. As the query requests become more frequent, the interrupts become more frequent. Just after the peak, the interrupt overheads start to counteract the gains from distributing the agent KSs, thus causing a decrease in the speedup. This speedup then starts to level off as more APs are connected because the next interrupt from the same agent KS will not occur until its current query is replied.

For all the different agent KSs set-ups, the peaks occur at even distribution or close to even distribution. Even distribution is when there is an equal number of agent KSs

running in each of the available AP. For ten, twelve, fourteen, and sixteen agent KSs set-ups, the peaks are at even distribution, i.e. with two agent KSs per AP. For eleven agent KSs set-up, the peak is at six APs which is close to even distribution (i.e. five APs with two agent KSs each and one AP with one agent KS). The peak for thirteen agent KSs set-up is at five APs with a value of about 3.36 but this is really close to six APs which has a value of about 3.35. The distribution of agent KSs for five APs is: two APs with two agent KSs each and three APs with three agent KSs each. The distribution of agent KSs for six APs is: one AP with three agent KSs and five APs with two agent KSs each. Therefore, the distribution of agent KSs at the peak for thirteen agent KSs set-up is really close to even distribution. The same is true for fifteen agent KSs set-up. The peak for fifteen agent KSs set-up is at six APs with a value of about 3.32, but this is really close to five APs which has a value of about 3.31. The distribution of agent KSs at the peak is: three APs with two agent KSs each and three APs with three agent KSs each. However, the distribution of agent KSs at five APs is even, i.e. three agent KSs per AP. Therefore, the distribution of agent KSs for fifteen agent KSs set-up at the peak is also really close to even distribution. The experiment in section 4.4.3 will discuss further about even distribution.

From thirteen agent KSs set-up onwards, there is a small increase in speedup at the end of the speedup curve. This slight increase at the end is due to the poor scheduling algorithm of the Linux kernel to handle large number of processes on a single processor [103]. This is also shown in the previous experiment and explained in section 4.2.3. This increase at the end can be seen more clearly as the number of agent KSs used in the set-up increases (e.g. in sixteen agent KSs set-up).

The efficiency results for all the agent KSs set-ups show a steady decline in efficiency as the number of APs increases. The efficiencies at the peak of the speedup for ten, eleven, twelve, fourteen, and sixteen agent KSs set-ups are about 71.8%, 55.0%, 57.5%, 48.1%, and 42.5% respectively. These efficiencies then end at about 31.8%, 28.2%, 26.4%, 21.6%, and 18.3% respectively. For thirteen agent KSs set-up, the efficiency at the peak speedup is about 67.2%, but at six APs (as the speedup value is very close to five APs), the efficiency is only about 55.8%. The efficiency then ends at about 24.4%. However, for fifteen agent KSs set-up, the efficiency at the peak speedup is about 55.4% but at five APs (as the speedup value is very close to six APs), the efficiency is about 66.2%. This efficiency then ends at about 20.0%. As can be seen the efficiency is inversely proportional to the speedup obtained. The efficiency's rate of drop decreases as the speedup's rate of climb increases.

The decrease in efficiency as the number of APs increases shows that the extra APs in the system is not being fully utilised. This is expected as only very specific parallel problems running on specific hardware can achieve and maintain full utilisation (i.e. efficiency of 1 as the hardware is scaled) [71]. The under-utilisation was observed to be due to waiting for a reply from the BB. This means that as the number of agent KSs per AP decreases, the less processing and more waiting each AP will do, thus reducing the efficiency. Detail tables of results for all the agent KSs set-ups in this experiment can be seen in Appendix F.

4.3.4 Conclusion

The general speedup trend as the number of APs increases is shown in Figure 42. In the beginning, the speedup increases quickly to a maximum point, after which it drops a bit

and slowly levels off. For larger number of APs (at the end), the speedup will start to increase slightly again.

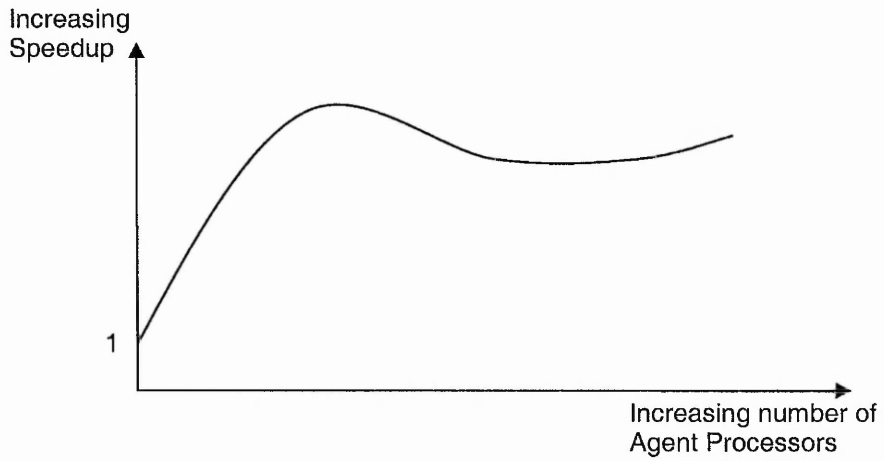


Figure 42. General speedup trend for increasing number of agent processors

The initial speedup increase is due to the reduced number of agent KS processes that each AP needs to time-slice between. This increase continues until the maximum point, after which the gains from distributing the processes are counteracted by the interrupt overheads and serial access to the blackboard. This means that the blackboard becomes the bottleneck for large number of APs. The small increase in speedup at the end (at large number of APs) is not due to the characteristics of the distributed blackboard system in the distributed processing network. This increase is caused by poor scheduling algorithm of the kernel for large number of processes [103]. Different operating systems with more efficient scheduling algorithms would most likely not show this increase.

In general, the distribution of agent KSs also needs to be even or close to even (for odd numbers of agent KSs) to be able to obtain an optimum speedup. The maximum speedup is generally obtained with two agent KSs per AP distribution. However, with fifteen

agent KSs set-up, the maximum speedup was obtained close to three agent KSs per AP distribution.

This shows that the maximum speedup is not only governed by the distribution of agent KSs but also the number of APs used. This is because large numbers of APs create large interrupt overheads for the blackboard thus reducing the speedup. These overheads saturate the BB and slow-down the BB's reply to requests from agent KSs. Therefore, to obtain maximum speedup for a given number of agent KSs, an even distribution of agent KS processes across the maximum number of APs that does not saturate the BB is required. The saturation point for the BB is when the BB cannot reply requests faster than the incoming requests rate.

Across the different number of agent KSs set-ups, the maximum speedup achieved is about 3.59 with five APs in the ten agent KSs set-up. This is still far from the suggested 5 or 10-fold speedup that is potentially possible [11]. This means that there is still plenty of room for optimising the agent KSs. For example, the structure of the rules in the agent KS can be rearrange to reduce the communication between the agent KS and the BB. The amount of data stored on the BB can also be reduced or compressed in order to reduce the amount of data sent per transmission. However, too little data stored on the BB defeats the purpose of using the blackboard architecture to share information between KSs. Therefore, a balance is required between storing information on the BB and the communication overheads. Another possible improvement is on the BB side, i.e. to enable concurrent access to different partitions on the BB. This would dramatically increase the maximum speedup. With these improvements it is possible to push the maximum speedup to nearly 10-fold [11]. However further gains in speedup will only be

possible if the parallelism granularity is finer, i.e. below KS process level. By reducing the granularity to many sub-KSs and for highly parallel problems it is possible to obtain speedup of up to 21-fold [6]. However, fine grain parallelism distracts the KS programmer with parallelism issues instead of concentrating on knowledge implementation.

Figure 43 shows the general efficiency trend as the number of APs increases. In general, the efficiency decreases slowly as the number of APs increases. This is because only very specific parallel problems on specific hardware can achieve and maintain efficiency of 1 as the hardware is scaled. Most general distributed processing would have some inefficiency. This inefficiency would accumulate as the number of processors increases. Thus, this is exactly what Figure 43 shows. This inefficiency is due to the under-utilisation of each AP. From observation, this under-utilisation is due to the idling time of each AP while it waits for a reply from the BB. As the number of APs increases, the less time each AP spends processing and the more time it spends idling waiting for the BB. Therefore the efficiency can be improved by processing more complex/processor intensive agent KSs.

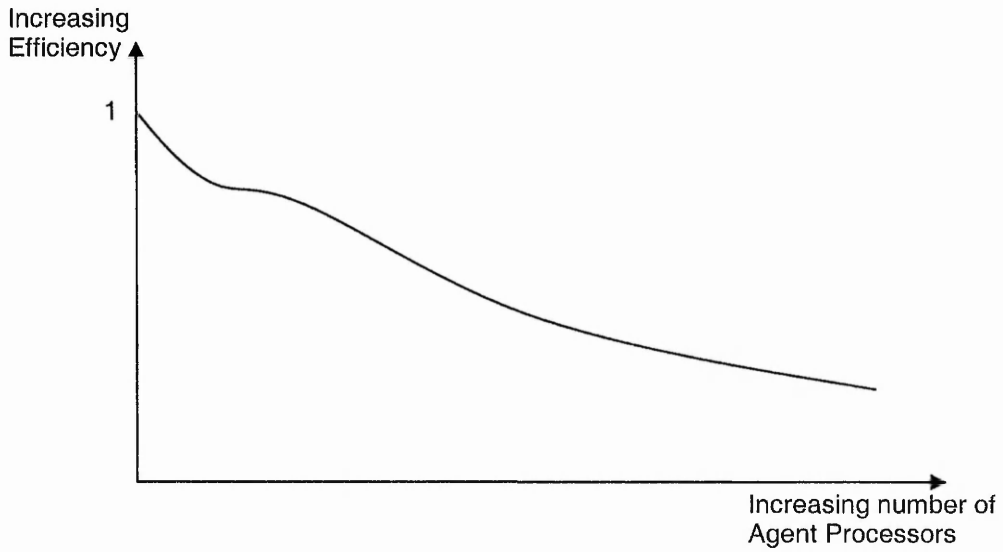


Figure 43. General efficiency trend for increasing number of agent processors

The efficiency trend also reflects the speedup obtained. The efficiency decreases slowly as the speedup reaches near the maximum point. After that, the efficiency starts to decrease steeply as the speedup starts to level off. Finally, the rate of decrease slows down as the speedup increases slightly at the end. In general, across all the agent KSs set-ups, the efficiencies at the maximum speedup point and at the last speedup point decrease as the number of agent KSs set-up increases. This is probably because the agent KSs are not processor intensive enough and do not make full use of the extra AP resources. More processor-intensive agent KSs would improve the efficiencies but another reason for the inefficiency is the saturation on the BB. Therefore, the efficiency (and therefore scalability) of a distributed blackboard system depends on the processor intensiveness of the KSs and the saturation point of the BB. To have good scalability, processor intensive KSs and high saturation point BB are required. The next experiment will investigate the optimum number of agents KSs to run for different numbers of APs with a given KS's processor intensiveness and BB saturation point.

4.4 Speedup and efficiency of varying number of agents

This is the third experiment carried out on TileWorld-DARBS. In this section, the aims and set-up of the experiment will first be explained. Then the results will be discussed and finally a conclusion of the results.

4.4.1 Aims of experiment

The aims of this experiment are to investigate the optimum number of agent KSs to run on a given number of APs and to evaluate its performance. Similar to the previous experiment, the performance is measured using speedup and efficiency. This experiment will also investigate the effects of limited processor resources on increasing number of agent KSs as would be the case in most distributed embedded processing networks running AI applications. The possibility of identifying the BB's saturation point and the processor intensiveness of the agent KSs will also be investigated in this experiment.

4.4.2 Experiment set-up

This experiment is run in eleven different set-ups. In the first set-up, one to sixteen agent KSs are run on one AP. The average time per move, speedup, and efficiency are calculated for each run. The second set-up is the same as the first except that two to sixteen agent KSs are run on two APs. The third set-up is also the same as the first set-up except that three to sixteen agent KSs are run on three APs. The same applies to the rest of the set-ups except that the starting number of agent KSs and number of APs used increases until eleven agent KSs running on eleven APs.

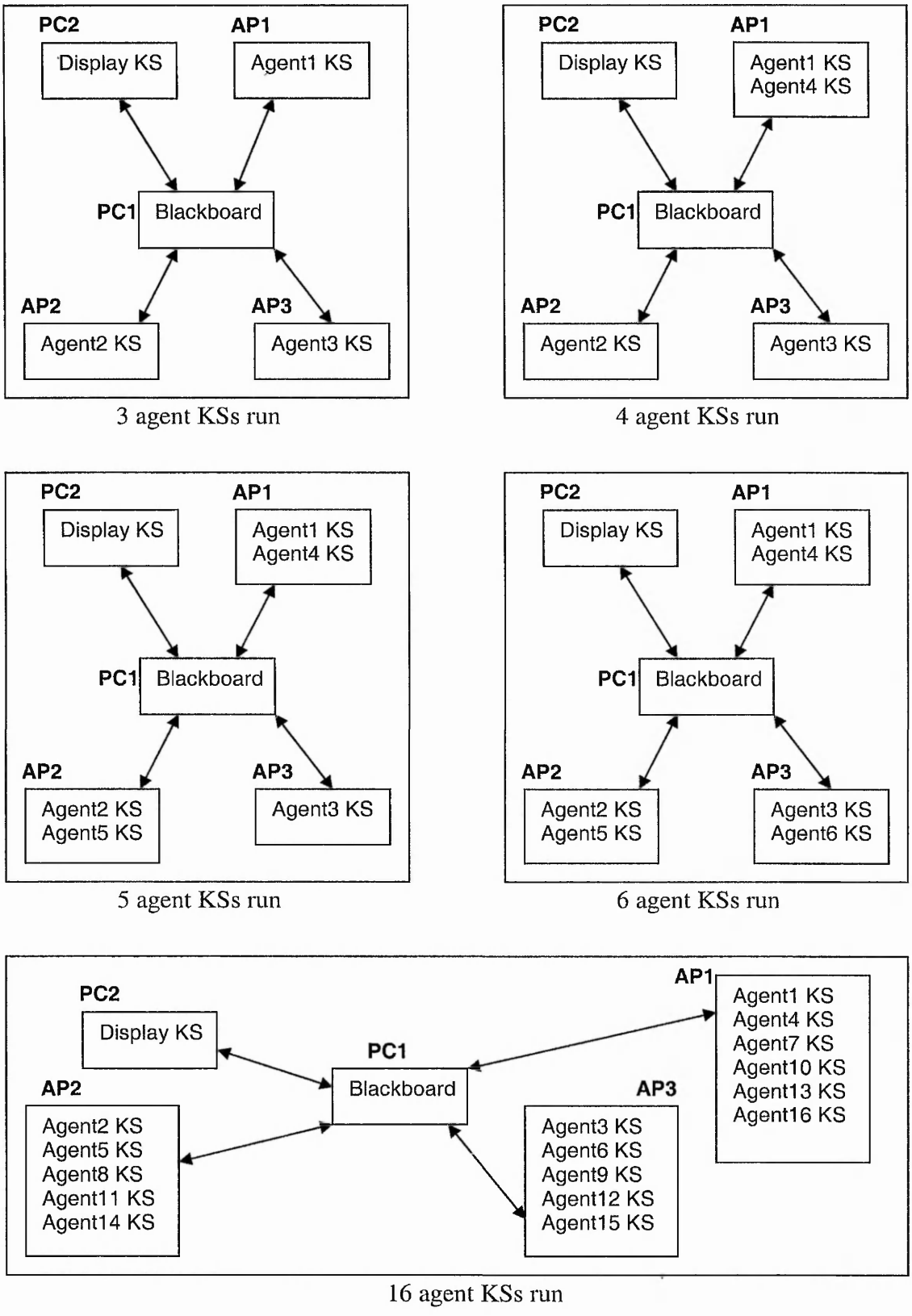


Figure 44. Three APs set-up with different number of agent KSs

Figure 44 shows an example of how three APs set-up with different number of agent KSs will run. As the number of agent KSs increases, they are distributed across the three APs as evenly as possible. The same applies for the rest of the set-ups.

4.4.3 Results and discussion

Figure 45 shows the speedup obtained for the different number of APs set-ups. Each APs set-up is colour coded as shown in the legend. A red dotted line is drawn to link the starting points of the different APs set-ups together.

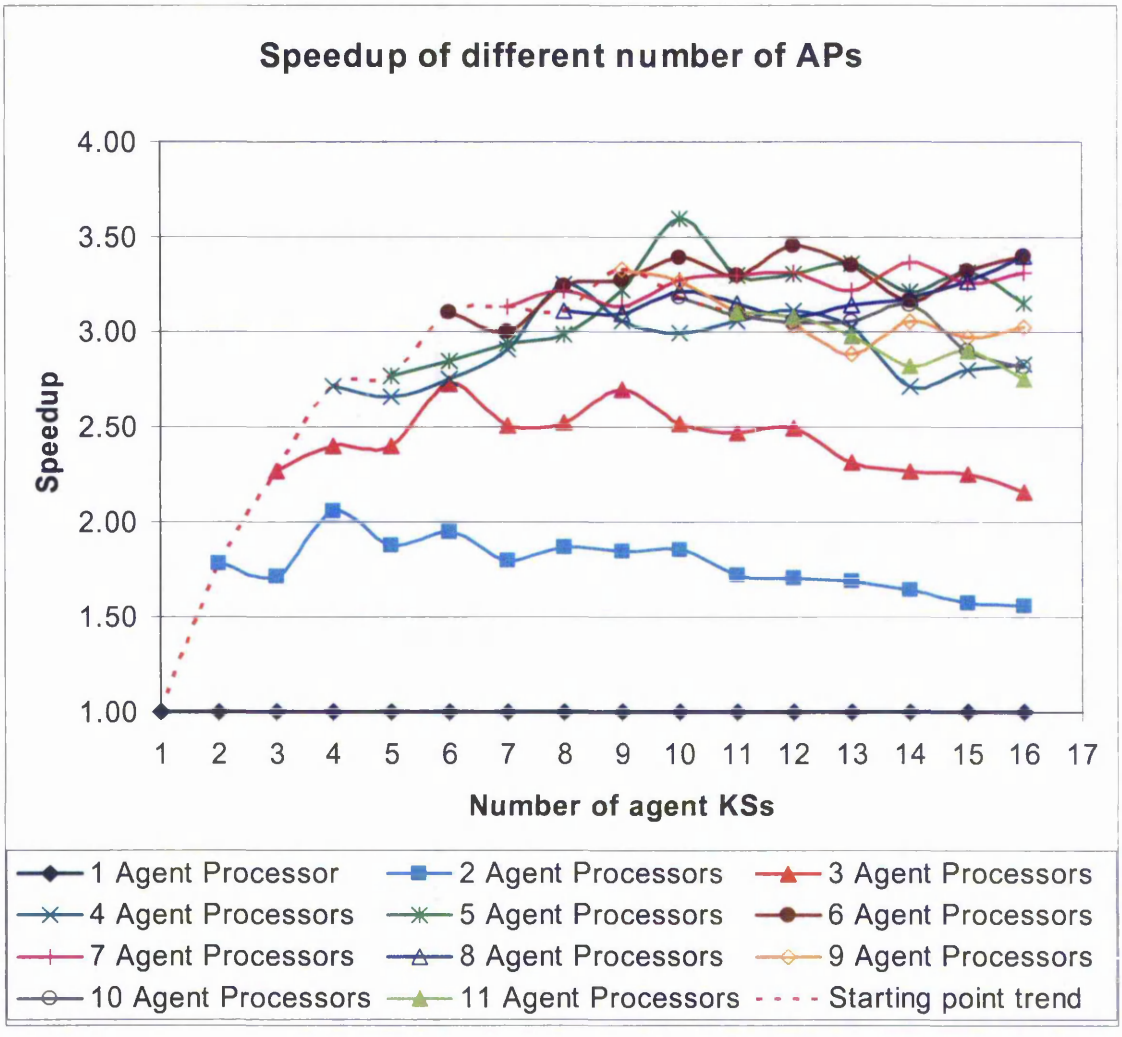
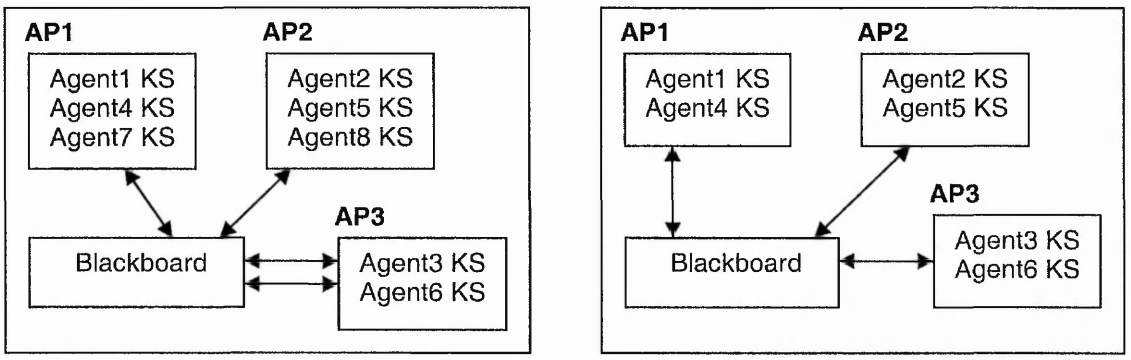


Figure 45. Speedup for different number of APs with varying number of agent KSs

The results in Figure 45 show that the speedup for each AP set-up peaks at multiples of the number of APs used. For example, two APs set-up peaks at four, six, eight and ten number of agent KSs whereas three APs set-up peaks at six, nine and twelve number of agent KSs. The reason for this was observed to be because when there is uneven distribution of agent KSs (Figure 46 (a)), the AP with the least agent KSs (AP3 in Figure 46 (a)) will be able to process faster, therefore, sending requests to the BB more frequently. This eventually hogs most of the BB's attention thus slowing down the BB's reply to the other agent KSs (AP1 and AP2 in Figure 46 (a)). In an even distribution (Figure 46 (b)), all APs have equal number of agent KSs to process, therefore, requests sent to the BB will have equal frequency.



(a) Uneven agent KS distribution with unequal access to the BB

(b) Even agent KS distribution with equal access to the BB

Figure 46. Example of equal and unequal access to the blackboard based on agent KS distribution

The peak at each multiple reduces as the number of agent KSs increases. This is because each AP needs to time-slice between more agent KSs as the number of agent KSs increases. The speedup eventually starts to decrease as the number of agent KSs per AP reaches four or five agents. This is because of the increase in context switching overhead.

For the two to eight APs set-ups, the highest peak achieved for each APs set-up is at two agent KSs per AP distribution. For the nine, ten and eleven APs set-ups, the results are

inconclusive as the maximum number of agent KSs used was sixteen. However, it is reasonable to speculate that the same would occur for the nine, ten and eleven APs set-ups.

The red dotted line shows the starting speedup trend as the number of APs set-up increases. The point when this trend line starts to cross the speedup of other APs set-up is in between seven to eight APs. At this point, further increase in number of APs does not guarantee further speedup. For example, the starting speedup for eight APs set-up is less than the speedup of eight agent KSs running on four APs. This is because by eight APs set-up, the BB is close to its saturation point. Therefore there is a higher speedup on less number of APs. However, the nine APs set-up is the only set-up where the start-up speedup is higher than its two KS per AP equivalent (i.e. five APs set-up with two KSs on four APs and one KS on one AP). Another reason for this is because the BB has only just started to reach its saturation point, therefore the BB's actual saturation point is around eight to nine APs. The reduced start-up speedup for the ten and eleven APs set-ups show clearly that the BB has gone beyond its saturation point. A detailed speedup results for all APs set-ups can be seen in Appendix G.

Figure 47 shows the efficiencies of the different numbers of APs set-ups. Each APs set-up is colour coded as shown in the legend. A red dotted line is drawn connecting the starting points of all APs set-ups. As can be seen, the efficiency of each APs set-up reflects the speedup obtained. The efficiency for each APs set-up reaches a peak at multiples of the number of APs used. The efficiency for four agent KSs running on two APs is about 1.03. This is most likely an error as the efficiency is not expected to exceed 1.00. This error can be explained to be due to the standard deviation and the low

resolution of the timer used. The resolution of the timer used was one second. The measured time for four agent KSs on one AP was 1 minute 55 seconds and on two APs was 56 seconds. Therefore if each of these measurements deviated by one second each, the results will be:

$$\frac{1:54\,\text{min}}{0:57\,\text{min}} = \underline{2.0 speedup}$$

$$\frac{2.0 speedup}{2 processor} = \underline{1.0 efficiency}$$

This means that four agent KSs running on two APs have maximum efficiency. However, the efficiency starts to drop as the number of agent KSs increases. This is due to the increase context switching overhead.

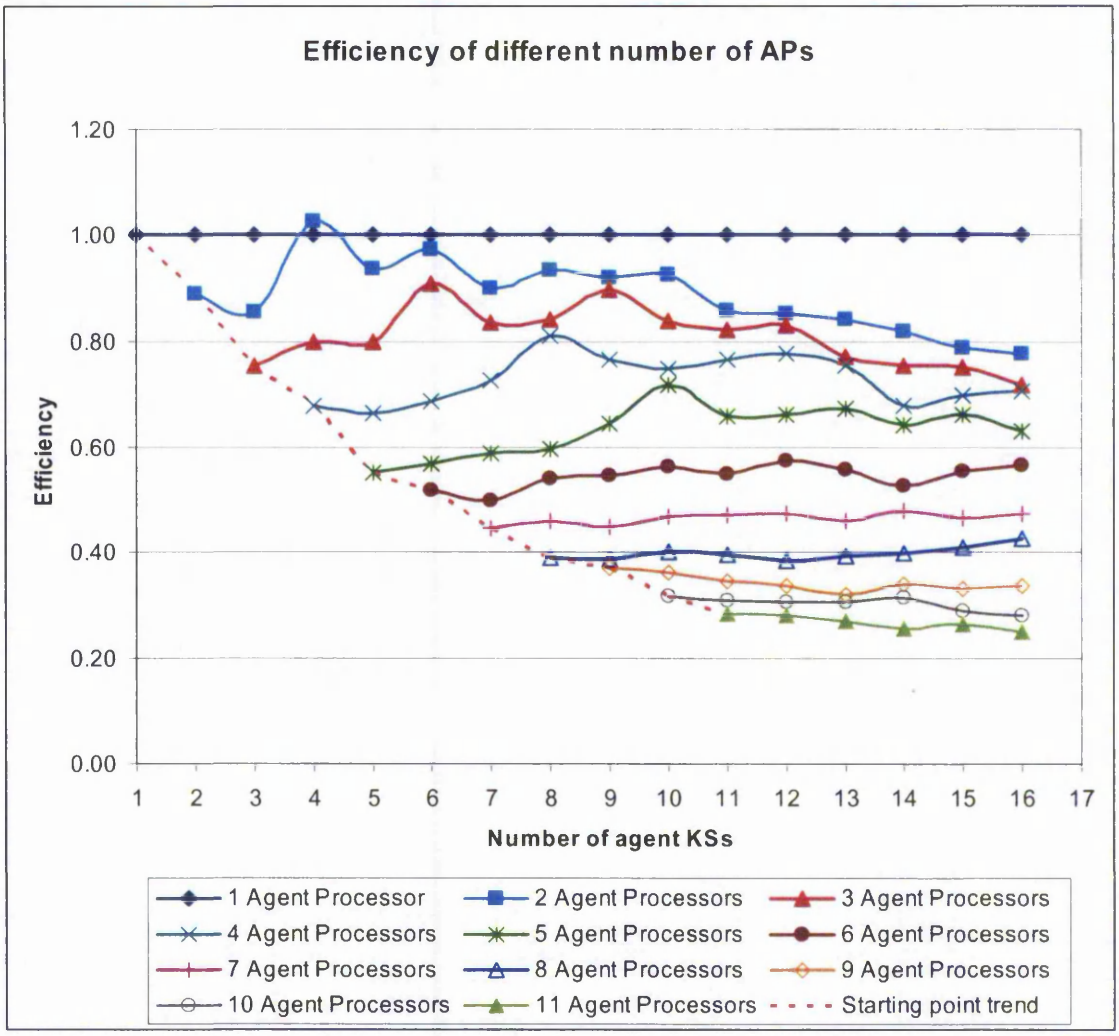


Figure 47. Efficiency for different number of APs with varying number of agent KSs

The efficiency also starts to decrease as the number of APs increases. This can be clearly seen with the red dotted line. This is expected because of the wasted processing resources. The extra processor added as the number of APs increases is not fully utilised by the current agent KSs. More processor-intensive agent KSs would provide better efficiency. It can also be seen that the efficiency for each APs set-up does not cross other APs set-up's efficiency. This means that in order to get the best efficiency for a given number of agent KSs of this processor intensiveness, it is best to use the smallest possible number of APs.

By the eight APs set-up, the efficiency line does not deviate greatly from their starting efficiency as the number of agent KSs increases. This is because by eight APs set-up, the BB's saturation point is close. This saturation point is the bottleneck for the performance. Therefore further increase in agent KSs will only show small if not zero efficiency gain. This is evident in the ten and eleven APs set-ups where the difference in efficiencies as the number of agent KSs increases is only about 0.02. A detailed table of efficiency results for all the APs set-ups can be seen in Appendix G.

4.4.4 Conclusion

From this experiment, the optimum number of agent KSs to run for a given number of APs is two agent KSs per AP. This will give maximum speedup and efficiency for this given number of APs. The speedup will peak at multiples of the given number of APs. However, each peak will be less than the previous peak until four to five agent KSs per AP. At this point further increase in agent KSs will decrease the speedup. This is because of the increase in context switching overhead.

From the speedup results in general, it can be seen that the speedup will initially peak at multiples of the number of processors used but will gradually decline as the number of KSs increases. This means that for limited processor resources, it is better to have the smallest possible multiple numbers of agent KSs. For example, if there is a limit of three APs, then the best number of agent KSs to run will be six.

The processor intensiveness for the agent KSs cannot be clearly identified in this experiment but from the efficiency results, it can be seen that the processor intensiveness

of two agent KSs can fully utilised the processing power of one AP. Further experiments will be required to clearly identify the processor intensiveness of these agent KSs. The saturation point of the BB was identified to begin at eight APs as it is the first speedup starting point that was not the highest speedup (see Figure 45). This saturation point can be identified on other similar distributed blackboard systems by running similar experiments to plot the starting speedup points of the different APs set-ups. The saturation point would be around the area when the starting speedup trend line starts to level out. This can be confirmed by checking other lower number APs set-ups running the corresponding number of KSs.

In general, the efficiency decreases as the number of APs increases. This is because the current agent KSs do not fully utilise the extra processor power available. More processor-intensive agent KSs would provide better efficiency. The efficiency also decreases as the number of agent KSs increases. This is because of the increase in context switching overhead. In general, the best efficiency is obtained by running on minimum number of APs with evenly distributed agent KSs.

4.5 Summary

The general aim of the experiments carried out on TileWorld-DARBS was to investigate the suitability, potential and characteristics of distributed blackboard systems on distributed processing networks. The results of these experiments would be the basis for determining the suitability of distributed blackboard systems in distributed embedded processing networks.

The first experiment carried out on TileWorld-DARBS proved that there are significant performance improvements running on distributed processing networks compared to running on a single processor. Although both distributed and non-distributed set-ups did not perform close to their respective ideal cases, the distributed set-up was far closer to its ideal case. For the non-distributed set-up, the overhead of context switching was identified as the main cause of slow-down. For the distributed set-up, an assumption of unlimited processor resources was made. However, most practical engineering applications will have limited processor resources. Therefore, the second experiment was carried out to investigate the speedup and efficiency of sharing the KS processes among varying number of processors.

The second experiment shows that the speedup increases rapidly as the number of APs increases up to a maximum point after which the speedup slowly levels off. The initial increase is due to the lesser number of agent KS processes that each AP needs to time slice between. After the maximum point, the interrupt overheads and the serial access to the BB start to counter the gains from distributing the processes. This decreases the speedup until a point where it levels off. This is basically the maximum speedup level of the BB as all the agent KSs will spend most of their time waiting on the BB. In general, good speedup is obtained with an even or close to even (for odd numbers of agent KSs) distribution, i.e. an equal amount of agent KSs running on each AP. The maximum speedup was observed to be influenced by the distribution of agent KSs and the amount of APs used. Therefore, to obtain maximum speedup for a given number of agent KSs, an even distribution of agent KS processes across the maximum number of APs that does not saturate the BB is required. The saturation point for the BB is when the BB cannot reply requests faster than the incoming requests rate.

The speedup results from the second experiment also show that it is far from the suggested 5 to 10-fold speedup [11]. This means that there is still plenty of room for optimising the KSs and the BB. One way of improving the BB's saturation point is to enable concurrent access to different partitions. In general, the efficiency decreases as the number of APs increases. This is mainly due to the idling time on the agent KSs while waiting for a reply from the BB. Therefore, the efficiency can be improved by having more processor-intensive KSs. From the second experiment, the scalability of a distributed blackboard system is influenced by the processor-intensiveness of the KSs and the saturation point of the BB. To have good scalability, processor-intensive KSs and high saturation point BBs are required.

The third experiment was carried out to find the optimum number of agent KSs to run for a given number of APs. It was found for TileWorld-DARBS that evenly distributing two agent KSs per AP gave the maximum speedup and efficiency. The peaks in the speedup and efficiency were found to be at multiples of the number of APs used. For example, with two APs the peaks are at four, six and eight number of agent KSs. The peaks decrease as the number of agent KSs increases until about four to five agent KSs per AP at which point the context switching overhead starts to cover the gain. Therefore, to have good speedup for limited processor resources, it is better to have the smallest possible multiple numbers of agent KSs. For example, if there is a limit of four APs, then the best number of agent KSs to run will be eight. The saturation point of the BB for TileWorld-DARBS was identified to begin at eight APs. Similar experiment can be run to plot out the starting points of the different number of APs set-ups to find the saturation point of other similar distributed blackboard systems. The saturation point

would be around the area when the starting speedup trend line starts to level out. This can be confirmed by checking other lower number APs set-up running the corresponding number of KSs.

As a whole, the results of these experiments show that distributed blackboard systems show performance increases when run on a distributed processing network and as such, are suitable for implementation in a distributed embedded processing network. To summarise the experiments, the first experiment proved that a distributed blackboard system running on a distributed processing network has better performance than on a non-distributed system. The second experiment shows that the speedup and efficiency characteristics of a set number of KSs running on increasing numbers of processors is maximised for even distributions of KSs among the processors. Finally, the third experiment shows the speedup and efficiency characteristics of a set number of processors running an increasing number of KSs is maximised when there are two KSs per AP.

5. Implementing an embedded distributed blackboard system

The results from the experiments in chapter 4 show that distributed blackboard systems have potential in distributed embedded processing networks. Therefore, this chapter concentrates on implementing the distributed blackboard system in a distributed embedded processing network. The resulting blackboard system is called embedded distributed blackboard system. This implementation took about eighteen months to implement due to the complexity of programming the SARNet system and the fact that there were also many hidden bugs in the SARNUX operating system.

5.1 Aims and requirements

One of the aims of this research was to implement a distributed blackboard system on a distributed embedded processing network. The choice of distributed processing hardware is not important as long as the behaviour of the distributed blackboard system can be investigated on a truly parallel platform. The choice of distributed blackboard system was already selected to be DARBS in section 3.1.1. Another aim of this research is to explore the challenges that arise from implementing a distributed blackboard system in a distributed embedded processing network. The knowledge and experience gained from this will help guide similar implementations in the future.

The distributed embedded processing network to be chosen should have a communications link that is sufficiently fast so that it does not cause a communication bottleneck during high communications periods. This is especially important if this system is to be run in an embedded system. The distributed embedded processing

network chosen should also be scalable and suitable for embedded systems. There should be appropriate underlying support (e.g. operating system and debugging facility) for the development of the blackboard system. There should also be appropriate access to the hardware so as to permit modification to the hardware if required to suit the research needs.

5.2 Design

This section will first discuss about selecting a suitable distributed embedded processing network based on the requirements stated in section 5.1. After which, some technical background on the selected distributed embedded processing network will be explained. The embedded implementation of DARBS will be called emDARBS from here onwards. The final part of this section will explain the actual designs of emDARBS.

5.2.1 Selecting a distributed embedded processing network

The choice of a distributed embedded processing network is not important as the main aim of this research is to investigate the suitability, characteristics and potential of the distributed blackboard system. However, a brief review of distributed processing networks is appropriate. Commercial distributed processing networks that are currently available in the market include the nCUBE [73]. The nCUBE is a specific-purpose distributed embedded processing network designed for video streaming and editing. As such it is not suitable for implementing the distributed blackboard system. In general, commercial distributed embedded processing networks incur high cost and are very specific-purposed designed, therefore they are not considered. The more research-based distributed processing networks include Intel's iPSC/860 [75], GMD FIRST's

PowerMANNA [110], and transputers [78]. The iPSC/860 is extensively used in numerical intensive parallel and distributed processing research. It consists of a hypercube network of i860 processors. Unfortunately Intel has ceased support and production of the iPSC/860.

The PowerMANNA node is made up of dual PowerPC MPC620 processors. Each node can have up to 1Gbyte of DRAM (8 × 128Mbytes). PowerMANNA uses a hierarchy of 16 × 16 crossbars to form a network of PowerMANNA nodes. The communication link is a synchronous, byte-parallel, bi-directional point-to-point connection operating at 60MHz. As the communication link is synchronous and parallel, many physical connections are required for each communication link. This increases the transmission speed but also increases the chances of fault occurring on the transmission lines and increases the cost of the network as it scales up. The maximum electrical power consumption of a PowerMANNA node is 70 Watts which is unacceptable for embedded systems as generally, embedded systems have a electrical power consumption constraint.

Transputers were formerly used in distributed processing network research. There were two types of transputers, INMOS transputers and Meiko transputers. Research at the University of Edinburgh used the Meiko multi-transputer system [6]. On the other hand, the INMOS transputers have been used in distributed audio video streaming research [111]. Unfortunately both the INMOS transputers and Meiko transputers have ceased production [112]. Nottingham Trent University's Parallel Processing Research group has developed SARNode as a replacement for the transputers and have used the ICR C416 router to link the SARNodes together to form the SARNets [113].

As the SARNet is located within the same institution, full access to the SARNet hardware has been granted. This means that, if necessary, modifications to the SARNet hardware can be made to suit the requirements of the research. This is essential to being able to successfully implement DARBS on the SARNet. The SARNet also provides the necessary underlying support for application development such as a real-time multi-threaded operating system for the SARNode called SARNUX [114]. The operating system is written in ANSI C and DARBS is written in C++. This makes it easier to implement DARBS on the SARNet. The SARNode also provides a debug UART (Universal Asynchronous Receiver/Transmitter) port for allowing the application program to send out debug messages [113]. This debug UART port can be connected to a standard personal computer (PC) serial port and using a simple terminal program on the PC (e.g. Hyperterminal), the debugging messages from the SARNode can be read. The SARNet's research group is currently researching implementation of the whole SARNode in a single chip design (System-On-a-Chip) [115][116]. This would allow the SARNet to be implemented more easily in an embedded system. A successful implementation of DARBS on the SARNet would pave the way for future use of distributed blackboard system in intelligent embedded systems. Because of the suitability of the SARNet for the implementation of DARBS and the convenience of being able to use the SARNet, the distributed processing network that is chosen is the SARNet.

5.2.2 Technical background on SARNet

The SARNet has been developed by the Parallel Processing Research Group of Nottingham Trent University as a replacement for the transputers [113]. The SARNet consists of a network of SARNodes connected together via an ICR C416 router [79].

The SARNode is designed with the embedded system in mind and each SARNode consists of a low power consumption 32-bit 200MHz StrongARM SA-110 RISC processor [20], 8 megabytes SDRAM, a 32-bit timer, I/O and UART debug port, and an OS-Link (over sampling link) [79] communication module. The ICR C416 router uses OS-Link communication protocol that uses the wormhole routing strategy, which has a low communication overhead [81][117]. The SARNet can have many different switch network configurations, one of which is shown in Figure 10 (page 29). Each ICR C416 router can be connected to other routers using any one or more of its OS-Link channels to make up a bigger network. A PC can also be connected to the SARNet via an interface card [118] as shown in Figure 10.

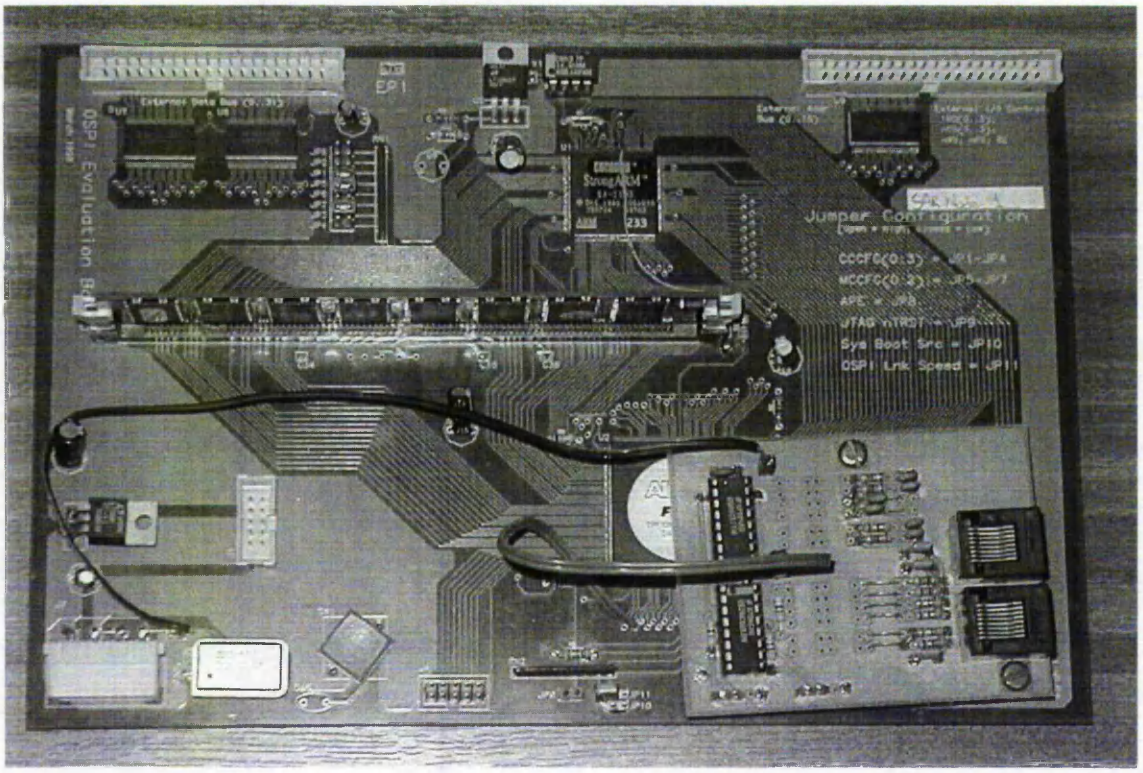


Figure 48. SARNode

Figure 48 shows a picture of a SARNode and Figure 49 shows a picture of the SARNet with four SARNodes connected together.

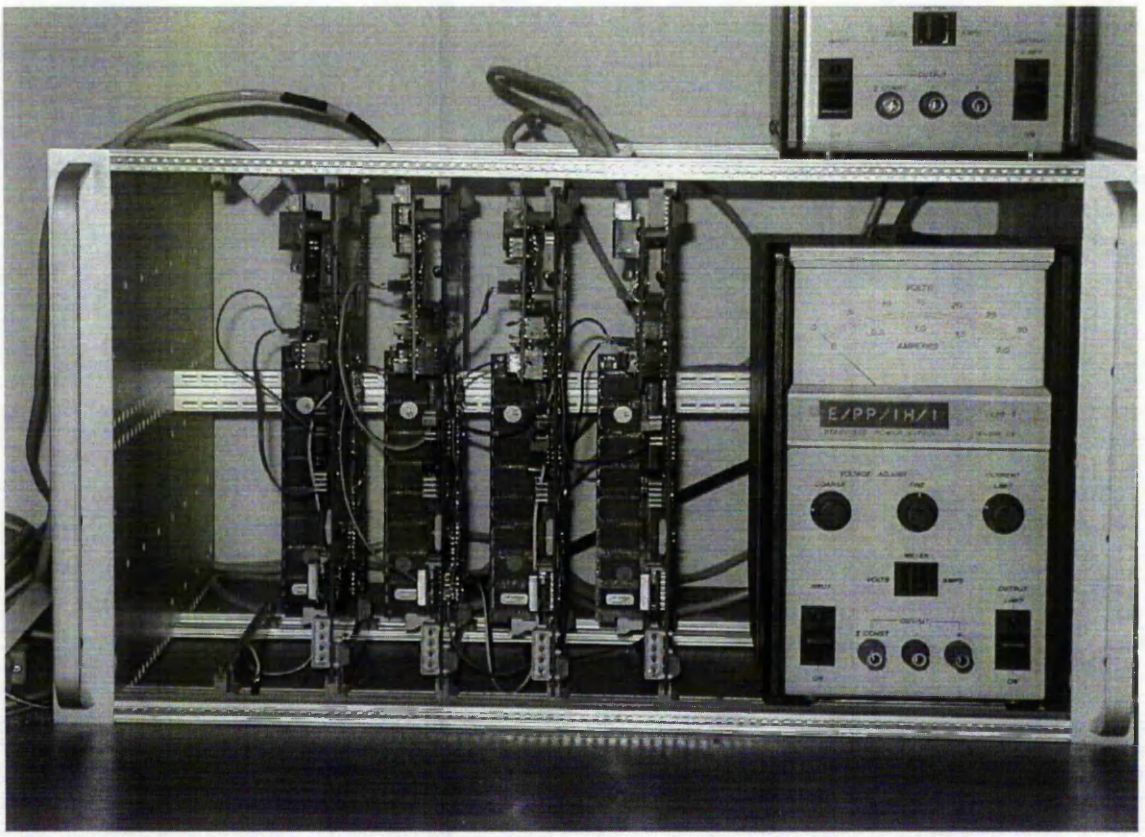


Figure 49. SARNet with four SARNodes

The operating system for the SARNode is SARNUX, which is designed for use in an embedded system [114]. The SARNUX is a pre-emptive real-time multi-thread operating system. It contains a process manager, transmitting manager, receiving manager, software channel manager, exception handler, timer manager, heap manager, semaphore manager, UART driver, and a stack walkback manager. The process manager handles process scheduling and queuing, while the transmitting and receiving managers provide an inter-process communication interface for application programs to use external OS-Link to communicate with other SARNodes. The software channel manager on the other hand provides the equivalent inter-process communication interface but for processes

within the same SARNode (i.e. internal communication). The exception handler handles interrupts and errors generated by the SARNode and the timer manager handles timer events and generates timer interrupts. The heap manager is used to manage the dynamic allocation and de-allocation of heap memory and the semaphore manager provides semaphores for concurrent processes to guard access to shared resources. The UART driver on the other hand provides low-level software support for sending out messages through the SARNode's UART debug port. Lastly, the stack walkback manager provides a means of using the debugging facility supported by Free Software Foundation's GNU project [119][120]. SARNUX has been successfully implemented in Quantel Ltd's distributed control system for real-time audio and video editing system (CLIPBOX) [121].

In SARNUX, the communication model is based on the communicating sequential process (CSP) model [122]. In the CSP model, the passing of messages between processes is through channels. The processes can be internal (within the same processor) or external (on a different processor). Each channel is a dedicated single direction communication link between two processes. If a bi-directional link is required between two processes then two separate channels would need to be declared. Communication between two processes would only take place when both the sender and the receiver of the channel are ready. Otherwise, the first process (either the sender or the receiver) would be blocked to wait for the other process to be ready. This is also used as a synchronisation method.

The SARNUX however has employed buffers in the external receiver link side of the communication channels to prevent link blocking. This is because the SARNode has two

physical OS-Link links and they operate on the send-and-acknowledge protocol [123]. If a process is not ready to receive and there is a message waiting for it to receive on the physical link then that physical link would be blocked. Other processes that are ready to receive or transmit cannot use that link. Thus, it can be seen that a deadlock can easily happen in this situation [124]. To prevent this, receiving buffers are used. However, if the receiving buffers are full then the link would be blocked. It is up to the application designer to make sure that this does not happen

5.2.3 emDARBS network layout and routing

The first thing that is required to implement emDARBS is to have an overall network layout. In order for communication to take place in SARNet, every message must contain a routing header and a unique message ID (MID). The routing header must contain the receiving port number, i.e. the destination SARNode's port number. The SARNet was designed to run as an embedded system and therefore all the network layout and settings are static and are known on compile time. The layout of the SARNet network for emDARBS to run on is as shown in Figure 50. The reason for choosing such a layout is to maximise the parallelism of the system as, at the moment, there are only four available SARNodes.

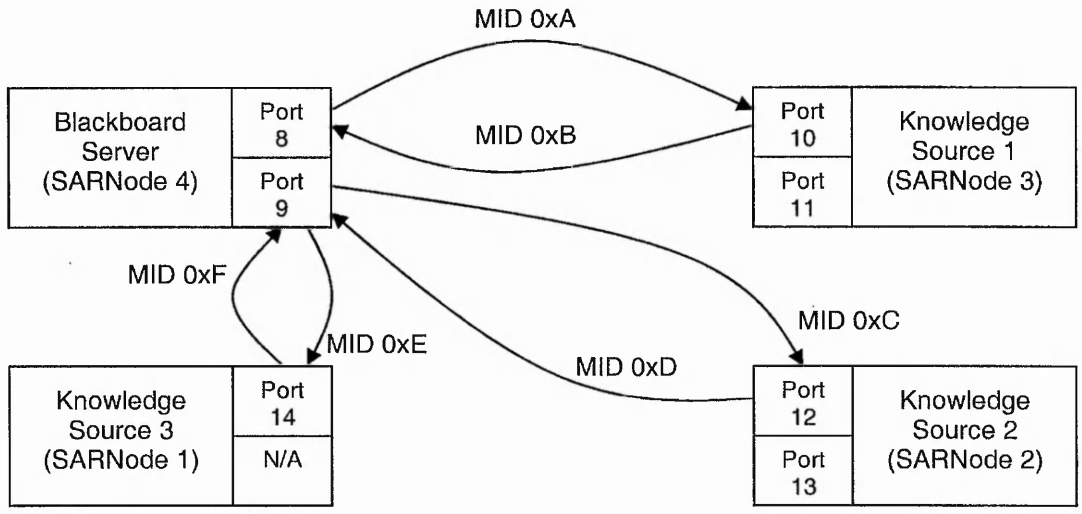


Figure 50. emDARBS network layout

The port number associated with each SARNode is also shown in Figure 50 along with the message ID (MID) used for each communication channel. The MID is only required to be a unique number for each communication channel and is therefore arbitrarily chosen to start from 10, i.e. 0xA in hex. The port number of the sender of a message is not important as the transmitting port is assigned dynamically. This means that every message could originate from either one of the two ports (except KS 3, which has only one port 14) available on the SARNode. For simplicity, Figure 50 only shows one of the ports that could originate a message. So for example, the BB server can transmit a message to KS1 via its transmitting channel using MID 0xA (in hex) to KS1's port 10. This message can originate from either port 8 or port 9 depending on which port is available at that time.

5.2.4 emDARBS inter-process communication model

After selecting the overall network layout for emDARBS, the next task to do was to design an inter-process communication (IPC) model. As already mentioned, DARBS

uses Linux's IPC model that uses the TCP/IP communication protocol (see section 3.2.1.2) while the SARNet uses SARNUX's CSP based communication model that uses the OS-Link communication protocol (see section 5.2.2). These communication models are not directly compatible. Therefore, an interface layer is required. From here onwards, references to TCP/IP communication protocol will refer to the Linux's IPC model that uses TCP/IP communication protocol and references to OS-Link communication protocol will refer to SARNUX's CSP based communication model that uses OS-Link communication protocol. DARBS already has a communication module and the core KS client and BB server are written to use the function calls provided by the communication module (see Figure 14, page 44). Therefore, to reduce the amount of changes required to the emDARBS code and to provide portability, the interface layer should provide the same function calls as the original DARBS communication module.

To do this, four new communication classes (two for the server side and two for the client side) were introduced. These classes act as wrapper classes for the original DARBS communication classes. Hence, no changes to the core DARBS are required, just the communication classes need to be redesigned. Thus, the new IPC model for emDARBS is as shown in Figure 51.

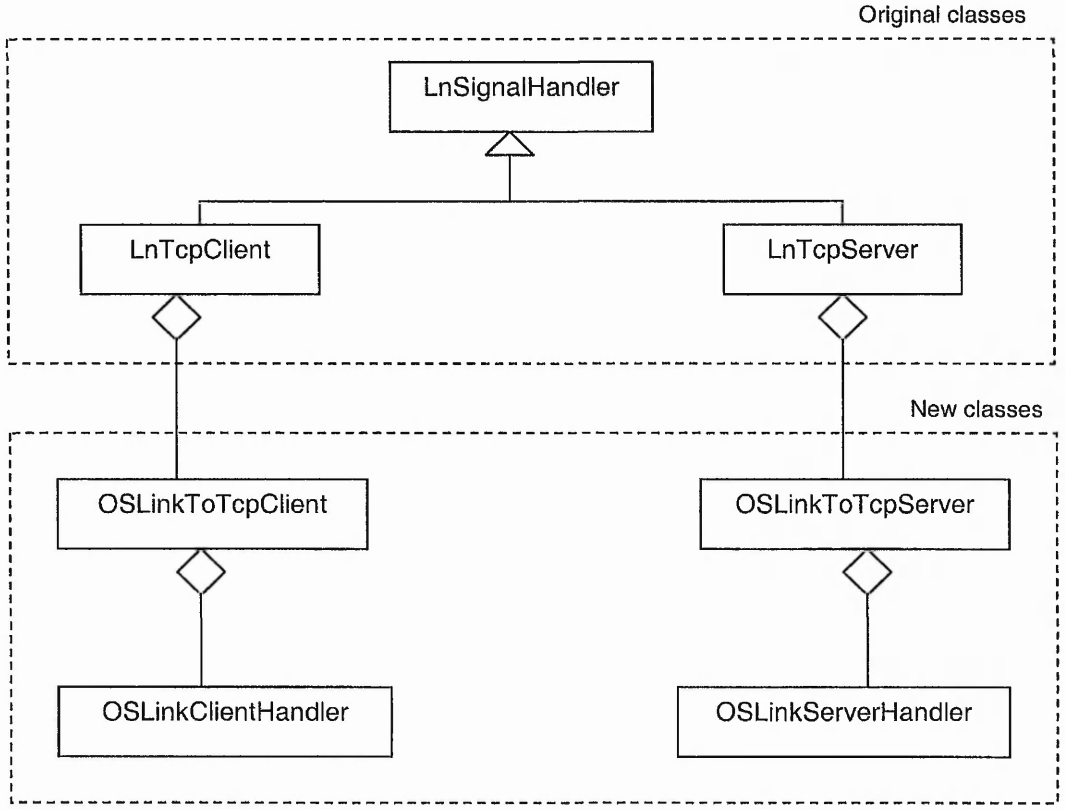


Figure 51. Classes in emDARBS IPC

As can be seen in Figure 51, the four new classes are: `OSLinkToTcpClient` and `OSLinkClientHandler` for the client side and `OSLinkToTcpServer` and `OSLinkServerHandler` for the server side. Since the original DARBS communication classes are replaced by wrapper classes, the functions of the original communication classes would be there but those functions would now be redirected to call the equivalent functions in the OS-Link communication protocol. This will be handled by `OSLinkToTcpClient` and `OSLinkToTcpServer` classes respectively. These two classes will then call the actual lower level OS-Link communication functions via `OSLinkClientHandler` and `OSLinkServerHandler` classes respectively. In this case, `OSLinkClientHandler` and `OSLinkServerHandler` are the actual classes that deal with the OS-Link communication and the `OSLinkToTcpClient` and

`OSLinkToTcpServer` classes are the interface classes to the original `LnTcpClient` and `LnTcpServer` classes.

5.2.5 emDARBS client side IPC

From Figure 51, it can be seen that the client side of emDARBS IPC consists of the `LnTcpClient` class, `OSLinkToTcpClient` class and the `OSLinkClientHandler` class. The SARNUX provides communication functions that are of the block receive and block transmit types. This means that the thread that calls a receive or transmit function will be blocked until the other end is ready to transmit or receive. So in order for the main KS client program to go on running and still have the communication running in the background, separate threads for receiving and transmitting must be used. With multiple threads running, some form of coordination/communication is required between the threads. The design of the KS client side of emDARBS that solves this problem is shown in Figure 52.

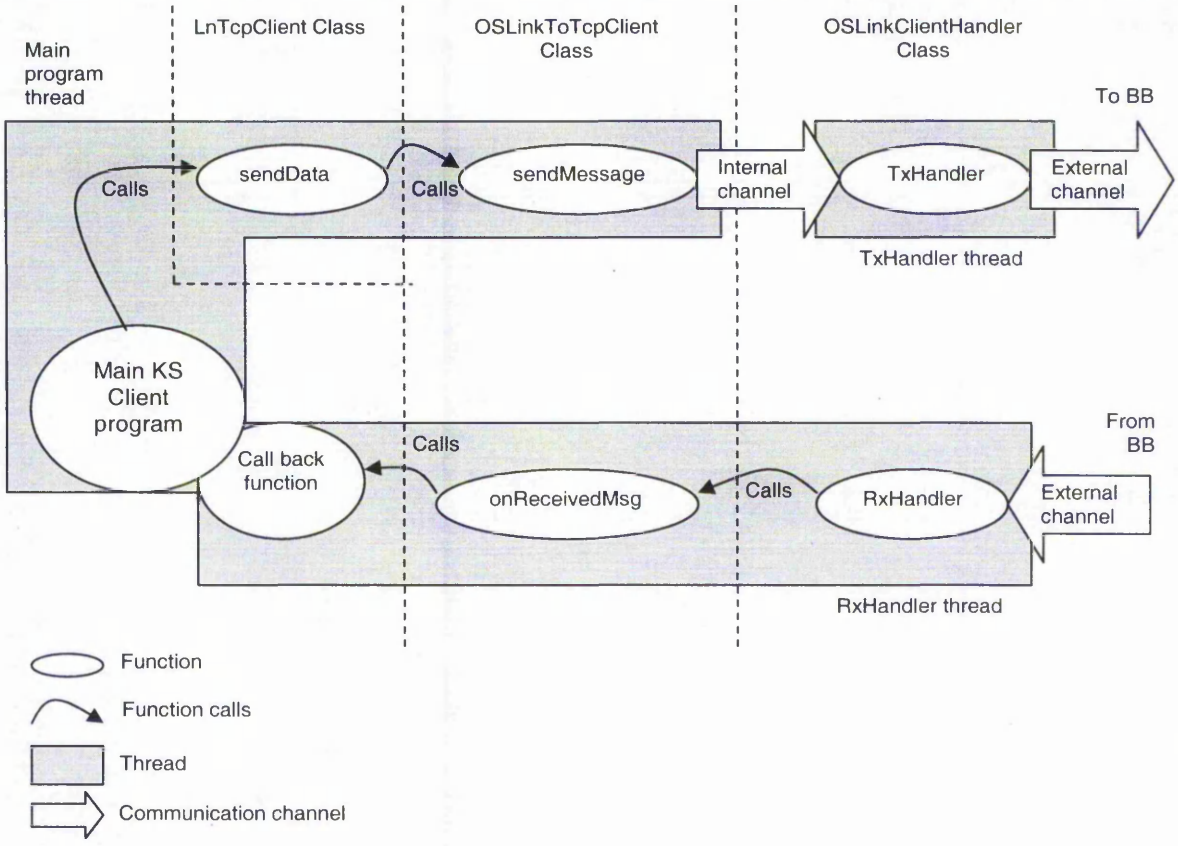


Figure 52. emDARBS IPC for KS client

As can be seen in Figure 52, there are three threads running concurrently: they are `TxHandler` thread, `RxHandler` thread, and the main KS client program thread. Communications between threads are accomplished via internal channels as can seen between `sendMessage()` function of main KS client program thread and `TxHandler` thread. The `RxHandler` thread would wait for a message to be received from the BB and then pass that message on to `onReceivedMsg()` function and then to the actual call back function of the main KS client. It is up to the main KS client and its call back function to arrange how they are going to pass information between them. One way would be to have shared variables and that the main KS client can monitor the shared variables for changes made by the call back function. The classes where each of

the function belongs can also be seen in Figure 52 and this gives an overview of how the layouts of the classes are in the client side of the emDARBS IPC.

5.2.6 emDARBS server side IPC

For the server side, the communication model is a bit more complicated than the client's IPC model as the server has to deal with multiple transmissions coming from different clients at the same time. The transmitting part of the server's IPC model will be explained first. Figure 53 shows the overview of the function of the transmitting side of the BB server.

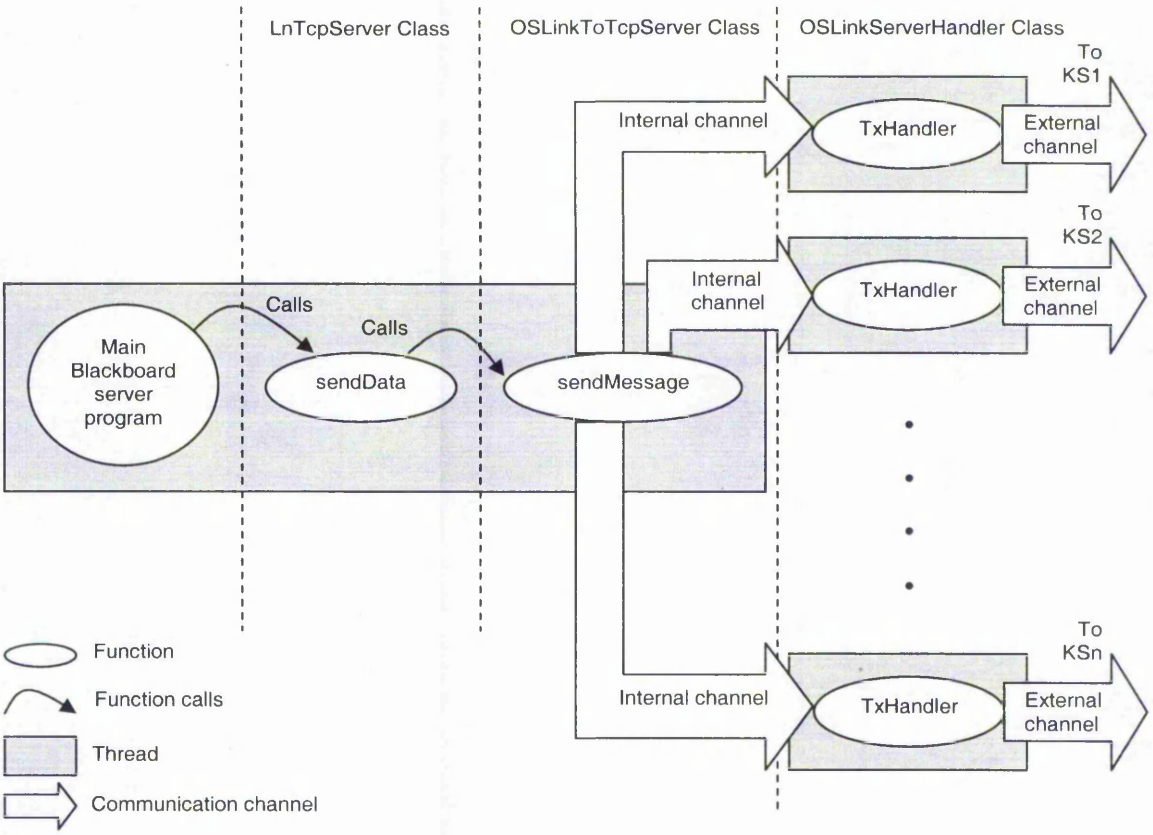


Figure 53. emDARBS IPC for transmitting side of BB server

As can be seen in Figure 53, there is a `TxHandler` thread dedicated for each KS client in the system. It is implemented this way so that KSs on the top cannot block or slow down KSs on the bottom from receiving a broadcast message. For example KS3 and below would not be blocked from receiving a broadcast message when KS2 is not ready to receive a message from the BB. The reason for this is that all the `TxHandler` threads are executed concurrently and are independent from the other `TxHandler` threads. Also shown in Figure 53 are the classes where each of the functions belongs.

For the receiver side, the server needs to make sure that the messages coming from each of the KS clients are not missed, and at the same time, has to make sure that the integrity of the data on the BB server are not compromised. To do this, access to the BB server has to be mutually exclusive [125], meaning that only one of the KS clients can access the BB server at any one time. Figure 54 shows how this is accomplished in the emDARBS IPC model.

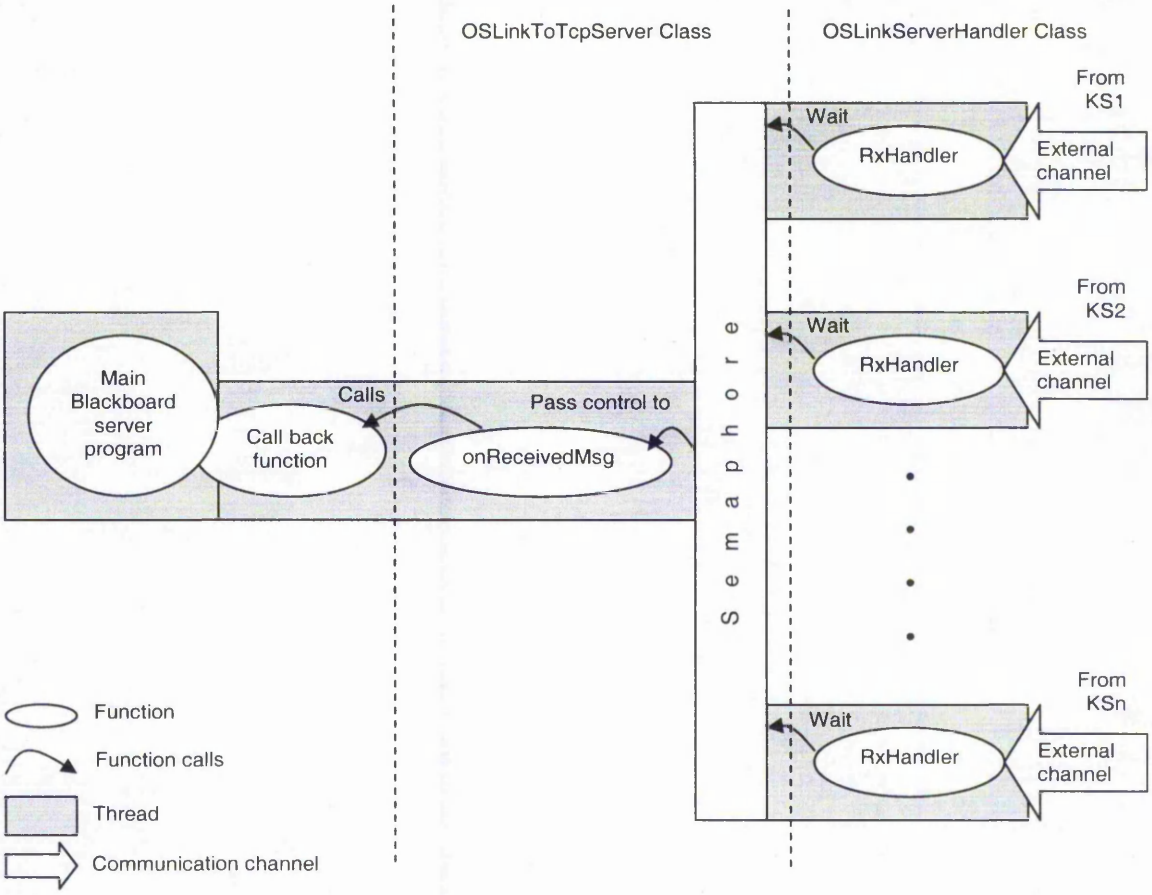


Figure 54. emDARBS IPC for receiving side of BB server

As can be seen in Figure 54, there is a `RxHandler` thread allocated for each KS in the system. Each one of the `RxHandler` threads is dedicated to monitor messages coming from its own KS client. When a message is received, the `RxHandler` thread then tries to wait on the semaphore [125]. If there is another thread already accessing the critical area (in this case `onReceivedMsg()` function) then the `RxHandler` will have to wait until the thread using the critical area has signalled the semaphore. Once the thread has access to the `onReceivedMsg()` function, the other `RxHandler` threads are blocked from accessing the `onReceivedMsg()` function until the thread has finished accessing the function. The rest of the function from `onReceivedMsg()` function onwards is similar to the client's receiver side. By using the semaphore on the receiver

side, mutual exclusion access to the BB is guaranteed and at the same time, the multiple `RxHandler` threads that are executing guarantees that the BB server does not miss any messages coming from the KS clients.

5.2.7 emDARBS external communication

After designing the inter-process communication, there was another communication issue that needed to be tackled: the external communication issue. An overview of how the whole system is set-up is required to illustrate this external communication issue and this is shown in Figure 55.

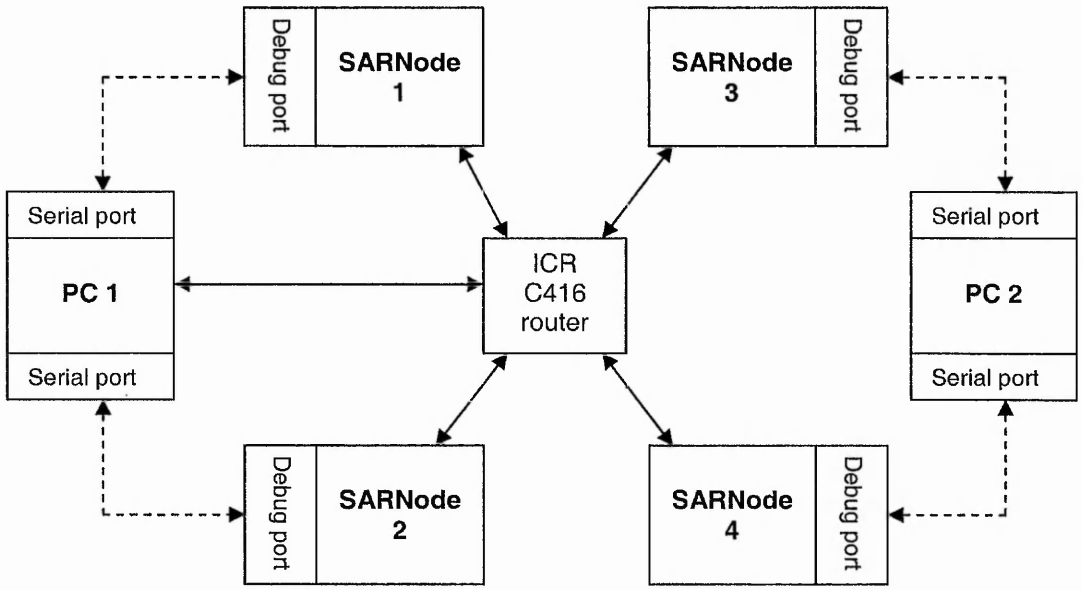


Figure 55. emDARBS system set-up

As can be seen in Figure 55, PC 1 is connected to the SARNet via the ICR C416 router. PC 1 starts the SARNet by downloading emDARBS onto the respective SARNode. Each SARNode has a UART debug port that is connected to one of the two available serial ports on a PC. This is for debugging purposes to show how emDARBS is running. The original DARBS KS client loads up files to determine what type of KS it is and to load

data to the blackboard. SARNUX, being designed for an embedded system, does not have a file structure and therefore needs a way to download this file to emDARBS at the start up of emDARBS. There are two ways of accomplishing this; one is to download the file via the UART debug port and the other is to download the file via the OS-Link. The speed of the UART debug port is 9600 bps and the OS-Link is 10 or 20Mbits/s. Therefore, after considering the size of the file to be downloaded, the speed of the communication link, and the purpose of the link, it was decided to use the OS-Link as the means of downloading the files to emDARBS.

To use the OS-Link for the download of the file would require the addition of a new external communication class in SARNUX. This class is called `ExternalComms` and it will be used by emDARBS to receive the file via the OS-Link. As there are two types of files to be downloaded (*.dkf and *.drf) an arbitrary choice of MID 0x5 for *.dkf files and MID 0x6 for *.drf files are used.

5.3 Implementation

A detailed list of all emDARBS IPC classes and its corresponding functions is in Appendix H. The full source code of emDARBS is in the CD-ROM attached to the back of this thesis in Appendix J. The rest of this section describes the challenges that were experienced during the implementation of emDARBS.

5.3.1 Challenges encountered

During the implementation of emDARBS, numerous challenging problems were encountered. These challenges were not encountered when DARBS was implemented on

a PC. Instead, these challenges are more specific to implementing DARBS in an embedded system environment. The first challenge was the cross-compiler. The original cross-compiler that was used to compile the SARNUX operating system was too old and does not support the new C++ STL (Standard Template Library) [126]. This meant that a new C++ cross-compiler was required. The other challenge encountered was that the SARNUX operating system was not 100% tested as is it not possible for a single developer (a previous research student) to fully test a complex operating system like the SARNUX operating system. Therefore, bugs in SARNUX were occasionally encountered and needed to be fixed. The following sections describe each of these challenges and explain how these challenges were solved.

5.3.1.1 Building cross compiler

The SARNode uses the StrongARM RISC processor which has a completely different instruction set than the normal PC-based processor. The programmer's editing environment is on a PC while the actual program has to be run in the StrongARM RISC processor which means that a cross-compiler is required. A cross-compiler is a compiler that runs on one type of machine (in this case the PC-based processor) and compiles the program for another type of machine (in this case the StrongARM RISC processor). The original C++ cross-compiler that was used to compile the SARNUX operating system is old and does not support the new C++ STL [126]. DARBS extensively uses STL therefore a new C++ cross-compiler was required.

There are commercially available PC-based-to-StrongARM-processor C++ cross-compilers but they require some form of licensing fee. There are also open source C++ cross-compilers that are available freely but because they are only available in source

code format, manually building the cross-compiler is required. In keeping with the future intention of being able to release the DARBS software as open source, the open source C++ cross-compiler was chosen as the cross-compiler. Another reason for choosing the open source C++ cross-compiler was to make sure that the software developed is free from any proprietary feature of the commercial C++ cross-compiler. This will make it easier for other future developers to pick up emDARBS or SARNet to use with their system.

In building the C++ cross-compiler, a certain sequence of build steps need to be followed. At each step of the build, the correct switch options need to be specified. Getting this sequence and switch options correct proved to be a challenging task as finding full up-to-date documentation of the open source cross-compiler is difficult. The low level functions of the cross-compiler also needed modification to suit SARNUX running on the SARNode. Emails were sent to cross-compiler newsgroups to seek advice from peers who had built cross-compilers before.

After about six months of trial and error and research reading on the Internet, the C++ cross-compiler was built and tested with the SARNUX test program. After successfully testing the cross-compiler, additional low level functions were developed to help support emDARBS such as the `cout` and `cin` functions. Because the SARNode is just a processor node and does not have any display monitor attached to it, the `cout` and `cin` functions were deemed unnecessary in SARNUX. However, because DARBS extensively uses `cout` and `cin` functions for debugging purpose, the `cout` function was implemented as writing output through the SARNode's UART debug port and the

`cin` function was implemented as reading information from the SARNode's UART debug port.

5.3.1.2 Debugging in SARNUX and emDARBS

As emDARBS was being implemented on the SARNUX operating system in small sections, bugs in SARNUX started to appear. This was expected as it was not possible to 100% test a complex operating system like SARNUX. In a typical embedded system, when a program crashes, especially when due to an operating system failure, it is not possible to break out of the program and debug the system. To overcome this, debugging messages were put into the code of the emDARBS program and the SARNUX operating system so that the last debugging message received could indicate roughly where the program crashed. Sections of the program code were also commented out and tested to slowly pin point the exact cause of the crash.

This technique is fine for simple bugs in the code but for complex bugs where the bug only appears when a sequence of events happen, this technique is not suitable. Instead sections of the code were ported back to the PC-based version and run using a PC-based debugger (GNU GDB [127] was used) to step through the program. This gave an insight of how the multi-threaded program was supposed to run. Often the bug would not appear in the PC-based version but because the PC-based debugger showed how the program should work, it helped to pin point where the bug might be in the SARNUX version of the program.

Another debugging tool that was available was the stack walkback feature that is supported by SARNUX. When compiling the source code of emDARBS, the debugging

option can be selected to put a table of the source code information (STAB) into the compiled executable. This would make the overall program very large but for the initial implementation stage, it is useful to have this type of information. When the program crashes with this table compiled in the executable, SARNUX would try to trace back the stack pointers to the program counter that causes the crash. SARNUX would then look it up in STAB to determine the exact line number of which file that caused the crash. This stack walkback feature is very useful provided that the crash is not fatal, i.e. the operating system can recover from it.

All in all, it took about twelve months to debug and fix the many bugs in the SARNUX operating system and the emDARBS implementation. Being multithreaded meant that most of the bugs only occurred with very specific sequence of events. Adding debugging statements to the code would change this sequence causing a different error to occur. Only through extensive trial and error were the bugs found and fixed.

5.4 Test and validation

After implementing emDARBS on the SARNet, tests were carried out to prove that emDARBS was working as it was designed to. The test plan was carried out in three phases. The first phase was to test that emDARBS IPC could send and receive messages. The second phase was to test the BB to make sure that it could handle multiple KS clients transmitting messages to it simultaneously. The third phase was to test that emDARBS could run a rule-based KS and perform the same functions as DARBS running the same rule-based KS.

For the first and second phase of the test, the original DARBS terminal program was used. The KS clients in the DARBS terminal program are just simple terminal clients. The KS clients take commands from the user and transmit them to the BB server. The BB server is the full running BB server used in DARBS.

For the first phase, one KS client and one BB server were run on two SARNodes respectively (SARNode 4 and SARNode 3 of Figure 50). The result from this test was positive as the new IPC functioned as it was designed to. A partition could be created and things could be added to the partition on the BB via the terminal client. Also the BB could transmit back to the terminal client the contents of the partition. This proved that the emDARBS IPC can transmit and receive messages.

In the first part of the second phase test, all four SARNodes were used, i.e. one BB server, and three terminal clients (SARNode 1, SARNode 2, SARNode 3, and SARNode 4 of Figure 50). Unfortunately, the clients crashed when the BB server was trying to broadcast a message to them. Investigation found that a deadlock occurred when the clients are trying to print out debugging messages on the UART debug port. This is because there are multiple threads running in a SARNode and each one of them are trying to access the single UART debug port concurrently. This problem was rectified by running the print out of debugging messages function with the interrupts masked. This guaranteed the print out of debugging messages function was mutually exclusive and therefore other threads would not be time-sliced in between access to the UART debug port. The test was rerun and this time the system worked. All the KS clients received the broadcasted message from the BB server correctly and could issue commands to the BB server correctly.

In the second part of the second phase, all three KS clients were programmed to automatically send requests to create a partition each and add 50 different data items to their partition. The KS clients were started off simultaneously and the BB server started to handle the multiple requests from the clients. The results of this test were compared to the expected results and found to be the same indicating that the BB server handled all the requests properly and as designed. This proved that the semaphore worked and that the data integrity of the BB server was not compromised.

In the third test phase, the *TestCompare KS* of the original DARBS was run on emDARBS. *TestCompare KS* is a simple rule-based KS that consists of two rules. The first rule is to add data sets to a partition and the second rule is to look through the data sets to find data that are more than a certain value and report them. emDARBS started up correctly and requested the KS file (`testcompare.dkf`) to be downloaded to the KS client via the OS-Link. `Testcompare.dkf` was sent to the KS client with MID 0x5 and was received correctly by the KS client (checked by the debug message coming from the UART debug port). Next the KS client requested the `setdatacompare.drf` rule file followed by `testcompare.drf` rule file. This was also sent to the KS client via the OS-Link (with MID 0x6) and checked using the UART debug port. As soon as the KS client received the last `testcompare.drf` rule file, the whole emDARBS started to work. Rules were created and fired. The test data were added to the BB and later compared for values that were more than 10 and reported them. The function of *TestCompare KS* in emDARBS was compared with the *TestCompare KS* in DARBS and deemed to be the same as they produced identical debug messages (a full listing of the `Testcompare.dkf` file,

`setdatacompare.drf` and `testcompare.drf` rule files, and the debug messages produced can be seen in Appendix I). This proved that emDARBS has been successfully implemented in SARNet, and that it is working as it was designed to.

After successfully testing emDARBS, the TileWorld test-bed application was ported over to emDARBS but due to memory constraint issues on the SARNet, this porting was not successful. Currently, each SARNode has 8 megabytes of memory. The current emDARBS implementation of the BB server occupies slightly more than 2 megabytes of memory space. The remaining 6 megabytes is then used for the OS-Link communication module's DMA engines, stack, page table, STAB information, and heap memory. As emDARBS uses STL [126], the heap memory consumption is relatively large. SARNUX also uses the heap memory to create control and stack space for each thread and because emDARBS extensively uses multiple threads at any one time, the heap memory consumption is also relatively large. The TileWorld-DARBS software running on the PC currently occupies about 7.1 megabytes of memory space but a large portion of this code is to do with graphical user interface. The emDARBS version would not have this graphical user interface code but even so, the remaining code generates large heap memory usage and as such is not possible to be implemented in the current SARNet. The next generation of the SARNet, XaNet is currently being developed which has 128 megabytes of memory capacity [116]. The XaNet will overcome the memory constraint issues on the SARNet and enable successful porting of the TileWorld test-bed application. Future works would include running the same performance experiments on the XaNet and comparing it with the TileWorld-DARBS results (see section 6.3).

5.5 Summary

One of the aims of this research was to implement a distributed blackboard system on a distributed embedded processing network. DARBS was selected as the distributed blackboard system and SARNet was selected as the distributed embedded processing network. DARBS uses Linux's IPC model that uses TCP/IP communication protocol (see section 3.2.1.2) while the SARNet uses SARNUX's CSP based communication model that uses OS-Link communication protocol (see section 5.2.2). These communication models are not directly compatible and therefore an interface IPC layer was designed and implemented to enable DARBS to run on the SARNet. During the implementation of emDARBS, many challenges were encountered. These challenges were not encountered when DARBS was implemented on a PC. Instead, these challenges are more specific to implementing DARBS in an embedded system environment. The two main challenges were the building of the cross-compiler and the debugging of SARNUX and emDARBS together. These two challenges took nearly eighteen months to overcome due to the complexity of programming the SARNet system and the limitations of the available debugging facilities.

After implementing emDARBS, tests were carried out in three phases to validate that emDARBS functions the same on the SARNet as DARBS functions on the PCs. The first phase was to test that emDARBS IPC could send and receive messages. The second phase was to test the BB to make sure that it could handle multiple KS clients transmitting messages to it simultaneously. The third phase was to test that emDARBS could run rule-based KS and perform the same functions as DARBS running the same rule-based KS. A deadlock problem was discovered during the tests and SARNUX was patched to correct this problem. On the whole, emDARBS passed all these tests and

proved that DARBS has been successfully implemented in a distributed embedded processing network. The TileWorld test-bed application was not successfully implemented on emDARBS due to memory constraint issues on the SARNet. It is planned for future work to implement TileWorld test-bed on the next generation of SARNet, XaNet which has larger memory capacity.

6. Discussion, conclusion and future work

This final chapter will first discuss and evaluate the work done in this research. It will then present a conclusion and, lastly, a discussion about possible future work.

6.1 Discussion

This section will discuss and evaluate the pros and cons of the TileWorld-DARBS implementation, the performance experiments carried out on TileWorld-DARBS, and the emDARBS implementation.

6.1.1 Pros and cons of TileWorld-DARBS

The TileWorld-DARBS implementation has shown that a distributed blackboard architecture is suitable for multi-agent systems. The individual agents can be suitably represented as an individual KS and, as DARBS does not have a control module, true opportunism and independence can be achieved by each *Agent KS*. The flexibility of DARBS also permits different implementation for each *Agent KS*. For example, there can be a neural network *Agent KS*, a genetic algorithm *Agent KS*, a rule-based *Agent KS*, and a conventional programmed *Agent KS* all working together through the BB. Mixed *Agent KS* implementations are ideal for solving complex real world problems.

The current implementation of TileWorld-DARBS uses rule-based *Agent KS*s. These agents are reactive agents [128]. Although reactive agents are more primitive and do not have forward planning, they are a lot easier to implement as rule-based KSs than cognitive agents [129]. The reactive agents in TileWorld-DARBS use random exploring

moves to guarantee that eventually, after some time, the entire TileWorld will be explored. This is not the most efficient way of exploring the entire TileWorld but it is the simplest. To explore the entire TileWorld more efficiently, cognitive agents with forward planning are required. Implementing cognitive agents as rule-based *Agent KS*s in DARBS is possible but it will require a lot more rules. The current inference engine used in DARBS is slow and inefficient at interpreting the rules. It breaks down each rule into all possible instantiations based on the number of variables in the rule. It also tests all the rules in sequential order and within each rule, checks each and every condition. These inefficiencies can be improved on as discussed in section 6.3.

The restart algorithm used in TileWorld-DARBS is also very primitive and inefficient. For the sake of ease of implementation, the restart algorithm is used to restart the KS every time a change in a relevant partition occurs. This is to make sure that the KS works with as up-to-date information as possible. This restart algorithm is easy to implement but causes the KS to slow-down as the number of changes to relevant partitions increases. Possible improvements to overcome this slow and inefficient restart algorithm will be discussed in section 6.3. However, even with improvements to the restart algorithm, there would still be some performance degradation. A better way to improve the performance is to change the agents' behaviour to cooperative and this is the subject of numerous agent behaviour research [104][105][106][107].

The objects in TileWorld-DARBS (i.e. the holes, obstacles, and tiles) are not dynamic. This can be argued to be not a complete TileWorld as the original TileWorld test-bed has dynamic objects. However, for the sake of simplicity, dynamic objects were not implemented. Dynamic objects are objects that can appear and disappear at different

places in the TileWorld over a period of time. This is to mimic the ever-changing real world environment. TileWorld-DARBS can easily implement dynamic objects by adding another KS that randomly moves the objects in the TileWorld. The rate of change for each type of object can be set by a user defined probability rate. This new KS will not require changes to be made to the other KSs as the restart algorithm will make sure that all relevant KSs start again whenever the *TileWorld Environment* partition changes. The only modification required is to add extra rules in the *Display TileWorld KS* to check for the movements of holes and obstacles.

As a whole, the TileWorld-DARBS implementation has resulted in improvements in DARBS, as one of the expectations when DARBS was selected for this research was to further improve it. DARBS has not been fully tested in a distributed processing network environment before and during the course of this implementation, faults were discovered when running DARBS in a distributed processing network. One such fault occurred during heavy communications when a deadlock occurred in one of the *Agent KS*. After further investigation, it was discovered that there was a fundamental flaw in the design of DARBS's inter-process communication model. This flaw would only show up during heavy communications and a detailed explanation of this can be found in section 3.4. This flaw was fixed by redesigning the inter-process communication model to use separate threads to service the incoming messages.

Another expectation for DARBS during the course of this research was to resolve any data inconsistency issues that arise from parallelising DARBS. The issues of data inconsistency were overcome by introducing the `replace` and `replace_multi` commands. These commands together with the unique information string format (see

section 3.2.2) made sure that the information on the BB is consistent. This is because the `replace` and `replace_multi` commands change the information on the BB in one atomic instruction. This guarantees that no other KSs can interrupt the BB while these changes are going on.

Finally, the flexibility of TileWorld-DARBS at adding and removing *Agent KS*s in the system shows that it is possible to use DARBS to implement redundant KSs. This is particularly useful in an embedded system that requires high reliability. Redundant KSs can be use to monitor the failure of a KS and automatically pick up where the original KS failed based on the information stored on the BB. This means that the information stored on the BB needs to be sufficient so that the redundant KS knows when a KS has failed and can easily pick up from where the original KS failed. However, a consideration needs to be taken when deciding the amount of information to store on the BB as storing too much information on the BB would result in a performance slow-down.

6.1.2 Pros and cons of the performance experiments

Detailed discussions of the individual experiments can be found in sections 4.2.3, 4.3.3, and 4.4.3 respectively. This section will only discuss about the overall pros and cons of all the experiments.

As a whole, there is a performance increase distributing TileWorld-DARBS across many processors but this increase is limited. The limited increase in performance is mainly due to the BB saturating. This is when the rate of requests to the BB is faster than the rate the BB can service the requests. There are two simple ways to overcome this: one is to increase the BB's processing power and the other is to reduce the amount of information

stored on the BB. However, having too little information on the BB defeats the purpose of a blackboard system. The purpose of having a blackboard system is so that the different KSs can make use of the information produced by other KSs to deduce newer information. This information may be useless to one KS but maybe useful to another KS later on. Therefore, there is a tendency to keep as much information as possible on the BB. Unfortunately, in distributed blackboard systems, too much information on the BB will result in performance degradation.

A more complex way of overcoming the BB saturation is to distribute the BB across a few processors. This has been done before in Knowledge Technologies International's NetGBB [62]. However, distributing the BB across different processors introduces more data consistency problems. NetGBB uses a control module and a defined blackboard language to maintain data consistency. DARBS on the other hand, does not have a control module and as such has more difficulties to maintain data consistency. There is a trade-off between central control KSs and fully autonomous KSs.

From the experiments, it was shown that distributing the KSs evenly across the available processors produces the best performance. This means that future implementation of intelligent embedded systems that use distributed blackboard systems need to have evenly distributed KSs across the available processors. If the available processors in the system have unequal processing power, then the most powerful processor in the system should be allocated to the BB and the rest of the processors should be evenly distributed among the KSs based on the processor's processing power. Another consideration when implementing distributed blackboard systems is to have a good balance of information that is stored on the BB.

The efficiency measurements from the experiments also showed that there is a considerable amount of idling time not utilised by the KSs of each processor. This further emphasizes the importance of choosing the right processor power for the task. As some embedded systems have processing power constraints, choosing the lowest processing power that can do the job is very important. Measuring the efficiency of each processor can show whether the processor is suitable for the job or not. These experiments have shown that running individual KSs does not require large processing power but having a group of KSs running together can require large processing power.

From the experiments, it was also noted that the measurements taken have a large standard deviation. This is especially prominent when large numbers of KSs are used. This is due to the poor restart algorithm used in TileWorld-DARBS. The restart algorithm was chosen for its ease of implementation and if TileWorld-DARBS was not used for performance test, this restart algorithm would be fine. However, this restart algorithm was used for performance tests and the result was a large standard deviation. A simple way of increasing the accuracy of the reading would be to take more samples and over more runs of the experiments. However, because each run of the experiments takes a very long time, this was not an option. Instead, the standard error of the mean is taken over a larger number of samples. This produced better measurements without the need to increase the number of runs for each experiment.

Unfortunately, the experiments carried out so far cannot really be generalised to all distributed blackboard systems as there are still many other criteria that need to be taken into account, for example, processor power and KS's processor intensiveness.

Experiments need to be carried out running TileWorld-DARBS on different processors to see the effects of lower and higher processing power. Experiments also need to be carried out to see the effects of KSs with different processor intensiveness on the overall system. Also more KSs on more processors need to be run to be able to plot the results better and to give a better trend line. Even with all these experiments, it is only possible to make some generalisations based on a distributed blackboard system with all identical KSs. It is quite impossible to run experiments with different KSs as there are endless combination possibilities. However, there may be a possibility to generalise the performance of a distributed blackboard system based on the processor intensiveness of the KSs.

6.1.3 Pros and cons of emDARBS

The successful implementation of emDARBS has proved that a distributed blackboard system with no centralised control module can be implemented in a distributed embedded processing network. During the course of this implementation, a software development tool-chain [130] which consists of an assembler, C/C++ compiler, linker, and downloader for the SARNUX was developed. This software development tool-chain is free from any proprietary features of commercial software development tool-chain and because it uses GNU General Public License [131], this tool-chain is free. This will greatly help future software development on the SARNUX and because it is free, it will also reduce the overall software development cost of embedded systems (ideal for cost constrained embedded systems).

The novel emDARBS IPC developed minimised the changes required to the core DARBS source code. This was proven in the testing phase of the emDARBS

development. This IPC can also be made more general and thus can be a standard interface for emulating Linux's communication model over SARNUX's CSP based communication model. With this general IPC, Linux based applications can be easily ported to SARNUX. However, on the down side, this type of emulating is not ideal for embedded systems as it introduces extra overheads, thus, slowing down the overall system performance. Ideally for embedded systems, the application program should be as efficient and small as possible. Unfortunately, there is a trade off between ease of portability and coding for efficiency. Future performance tests could be carried out to see if this extra overhead is still within the acceptable tolerance of an embedded system.

There is a memory constraint on the SARNode. As emDARBS was being developed, it was observed that the memory usage is quite large (slightly more than 2 megabytes for the BB server). Because SARNodes were developed for the embedded system the available memory capacity is only 8 megabytes per node. After splitting up the memory for the OS-Link communication module's DMA engines, stack, page table, and STAB information, only about 6 megabytes of memory capacity is left. The remaining 6 megabytes of memory is then shared between the application code and the heap memory space. As emDARBS uses STL [126], the heap memory consumption is relatively large. STL is useful for programming as it reduces the amount of coding required but it produces large executables and uses large amount of memory capacity. At the moment the basic emDARBS is still within the 8 megabytes memory space but any larger application would have a memory capacity issue. For example, the PC-based TileWorld-DARBS application currently occupies about 7.1 megabytes of memory space. A large portion of this code is to do with graphical user interface which emDARBS would not use but because the core code generates huge heap memory usage, TileWorld-DARBS

would not fit in the current version of SARNet. On the other hand, because emDARBS is ported from DARBS (which is not coded with memory capacity constraint in mind), there is a lot of redundant code in emDARBS. Therefore, removing this redundant code in emDARBS would help reduce the overall memory consumption and possibly speed up the overall program. Alternately, research work is being done on the next generation of SARNode which would be implemented on a single System-On-a-Chip [115] solution. This new generation of the SARNet is called XaNet and it consists of Excalibur processors which are based on the ARM922T architecture [116]. Each Excalibur processor has a UART, dual channel 32-bit timer, and 128 megabytes of SDRAM.

There were also weaknesses on both the SARNode and SARNUX in terms of its debugging facilities. The SARNUX operating system only uses the UART to transmit debugging information as the SARNode hardware only provides a UART port for debugging purposes. This is suitable for debugging small programs on the SARNUX but is not suitable for debugging large programs that run as multiple threads. The SARNUX operating system tries to compensate this by providing a stack walkback feature for tracing back to the crash point of an application. However, this is only good if the crash is not fatal and if the operating system is able to recover from the crash. It would be better if the SARNode and SARNUX supported program could be stepped through and runtime variables watched. The XaNet provides IEEE Standard 1149.1 (JTAG) [132] connector that can support runtime debugging. However, the supporting JTAG debugging software that is currently available is costly. There is GNU GPL JTAG debugging software but some work is required to configure and build it to work with the XaNet.

At the moment, emDARBS does not take into consideration real-time constraints such as guaranteeing a response in a given time frame. Some embedded systems have real-time constraints and in order for emDARBS to be used in future intelligent embedded systems, it needs to have features that take into account real-time constraints. emDARBS is currently too slow for many practical real-time embedded systems and one way of speeding it up is to strip down its code to use only Embedded C++ [133][134]. Embedded C++ is a subset of C++ that is specially catered for embedded systems. In Embedded C++, exception handling is not permitted and because DARBS uses exception handling, future version of emDARBS should remove all these exception handling. In the future, emDARBS should be a subset version of DARBS that is specially designed to cater for implementation in embedded systems.

As a whole, emDARBS is a good architecture for intelligent embedded systems as it provides the means for implementing distributed knowledge sources in a system. It is good for a complex real world environment where it is not possible to code for all the possible situations that can occur. Using knowledge sources, the general "common sense" knowledge can be coded into the system and different combination of the knowledge sources would then work out what to do based on the changing environment. This research has put emDARBS one step closer to the implementation of intelligent embedded systems.

6.2 Conclusion

The future of embedded systems lies in its intelligence. Future intelligent embedded systems will need suitable software and hardware architectures to support their complex artificial intelligence software. Distributed blackboard systems are suitable software

support architectures and distributed embedded processing networks are suitable hardware support architectures. DARBS is a distributed blackboard system developed by the Open University and Nottingham Trent University [59]. It was chosen as a suitable distributed blackboard system for this research because it does not contain a control module. This means that DARBS permits truly opportunistic KSs to be implemented. Another reason for choosing DARBS is because it is written in C++ which is a suitable language for embedded systems and full access to the source code is also available. The focus of this research was to investigate the suitability, potential and characteristics of a distributed blackboard system in a distributed embedded processing network. This has been accomplished by running performance experiments with the distributed blackboard system and implementing the distributed blackboard system in a distributed embedded processing network.

The TileWorld test-bed was selected as the application to run on DARBS for the performance experiments. The TileWorld test-bed is a well established multi-agent system test-bed developed by Pollack et al [87][88]. The TileWorld test-bed was implemented on DARBS with the agents implemented as KSs and the contents of the TileWorld stored on the BB. The *Agent KS*s were implemented as reactive agents that use randomly generated moves to explore the TileWorld. The TileWorld implementation on DARBS is called TileWorld-DARBS. During the course of implementing TileWorld-DARBS, some design flaws in DARBS were discovered. These flaws include the inter-process communication module flaw which causes a deadlock to occur during intensive communications. This flaw was rectified by changing the inter-process communication module to use threads to service incoming messages.

Performance experiments were carried out to investigate the suitability, potential and characteristics of distributed blackboard systems on distributed processing networks. The results for TileWorld-DARBS on distributed PCs show that there is a performance increase when distributing the processes across different processors but the performance increase is limited. This is mainly due to the BB's saturation. The performance can be increased further by changing to fine grain parallelism but this will distract the KS designer with fine grain parallelism issues instead of concentrating on knowledge implementation. Therefore, this research focused on parallelism at KS level granularity.

In general, the results of the performance experiments show that if the number of KSs is set, then increasing the number of processors will increase the performance up to a point where the communication frequency is too high (i.e. when the BB saturates and cannot handle the rate of incoming requests). After this point, any further increases in the number of processors will result in degradation or no improvement in performance. Conversely, if the number of processors is set, then evenly distributing the number of KSs among the available processors will generally increase the performance of the system (e.g. for three processors system, the peaks are at three, six, and nine KSs). However, this increase in performance will start to decline when the context-switching overhead is more than the gain in distributing the KSs. On the other hand, if both the number of processors and KSs are increased at the same time, then the performance increase is limited by the saturation of the BB. The experiments also show that distributed blackboard systems can be suitable architectures to implement multi-agent systems. These agents can be implemented on separate processors thus, making them truly autonomous and independent agents.

DARBS has also been implemented in a distributed embedded processing network. The distributed embedded processing network that was chosen is the SARNet [113]. The SARNet is a network of SARNodes developed by Nottingham Trent University's Parallel Processing Research group as a replacement for the transputers. The implemented DARBS on the SARNet is called emDARBS. During the implementation of emDARBS, many challenges were encountered. It was learnt from these challenges that building complex software for embedded systems requires a proper software development tool-chain.

The success of emDARBS shows that as a whole, emDARBS is suitable for future intelligent embedded systems, but more work is required to reduce the memory consumption of emDARBS. Work is also required to include real-time features in emDARBS. In general, the architecture of distributed blackboard systems enables easy implementation of redundant KSs as backups for the system. This is particularly useful for reliability constraint embedded systems. To tackle the problem of memory consumption, it is suggested that future versions of emDARBS are to be written in Embedded C++. Embedded C++ is a subset of the Standard C++ that is specially catered for embedded systems. With this, future versions of emDARBS will be a subset of DARBS that specially caters for embedded systems. Similar performance experiments could then be carried out to compare the difference in performance between the PC-based DARBS and the SARNet-based Embedded C++ emDARBS.

As for the SARNet, it is just sufficient for emDARBS, any increase in software complexity will cause the SARNet to run out of memory capacity. Therefore, it is suggested that the next generation of SARNet is to have bigger memory capacity. The

newly developed XaNet is the next generation of SARNet and it contains a 128 megabyte SDRAM. The SARNet also has a poor hardware support for advance software debugging. The XaNet has improved on this by adding hardware JTAG debugging support but some work is still required to develop the debugging software that uses the JTAG on the XaNet. As for the SARNUX operating system, it is good in terms of its memory footprint (the whole SARNUX executable only occupies about 64 kilobytes) [124], but it lacks advance debugging facilities. Future versions of the SARNUX would need to improve on its debugging facilities.

As mentioned earlier, the challenges encountered during the implementation of emDARBS showed that developing complex software for embedded systems requires a proper software development tool-chain that includes an advance debugging software. Without this tool-chain, it would be nearly impossible to develop complex software and trace down any error that is in the software. There are commercially available software development tool-chains but they usually require expensive licensing fees. Also, some tweaking is still required to customise such commercial tool-chains for the SARNet. Alternatively, open source software development tool-chains can be used but a lot of work is required to configure and build the tool-chain for the SARNet. However, open source software development tool-chain is free and once customised for the SARNet, it can be used freely without the constraint of licensing fees.

Distributed blackboard systems are feasible in practical embedded systems but the effectiveness of the system would greatly depend on how the KSs and the information stored on the BB are implemented. Poorly distributing the problem onto the KSs would result in an ineffective system due to poor performance. For example, distributing the

problem onto too many KSs will cause an increase in unnecessary communications while distributing the problem onto too few KSs will increase the processing load of each KS. Poorly organising the information stored on the BB would also result in poor performance. For example, storing too much information on the BB would increase the communication load, thus reducing the performance of the overall system. Conversely, storing too little information on the BB defeats the purpose of using a distributed blackboard system and makes it more difficult for redundant KSs to quickly pick up from where a KS has failed. Finally, distributed blackboard systems also need to have real-time capabilities for them to be feasible with real-time embedded systems.

6.3 Future work

As discussed in section 6.1.1, DARBS's rule-based system is currently inefficient and as such there are possible future works for improving the rule-based system. One such work involves implementing a rule dependency table [52]. As many rules in the system depend on the condition of other rules, it is therefore possible to build up a table of all these dependency before runtime and have the inference engine select rules to be tested based on this table. For example, if rule B cannot fire unless rule A has fired (i.e. rule B is dependent on rule A), then there is no point in checking rule B until rule A has fired. Another possible improvement to the rule-based system is to have runtime evaluation of the composite condition in a rule. The current inference engine checks every sub-condition of a rule before evaluating whether the composite condition is true. This can be improved on by evaluating the composite condition as it checks each sub-condition. Consider the following example in Figure 56.

```
IF
[
      condition1
      AND
      condition2
      AND
      condition3
]
THEN
[
      actions
]
```

Figure 56. KS rule example

The inference engine can stop checking *condition2* and *condition3* if *condition1* is false because the composite condition will be false. This type of runtime evaluating would reduce the number of messages sent to the BB as each condition-check requires the KS to send a message to the BB. The use of runtime evaluation can dramatically reduce the communication traffic, thus allowing other KSs to send and receive messages faster.

The current restart algorithm of the KSs is also relatively slow and inefficient. At the moment, each KS will restart on a change in any partition that is in its working partitions list. This is inefficient as most of the time KSs only need to restart on a subset of the working partitions list. Therefore, this can be improved by having KSs selecting particular partitions within the working partitions list to restart on. Another improvement to the restart algorithm is to have the KS check the BB for the specific changes and then decide whether the changes have affected the KS's current evaluation. If the changes do affect the current evaluation, then the KS should restart, otherwise, it can continue

evaluating its current evaluation. These improvements to the restart algorithm will provide some increase in performance.

Another possible future work for the *Agent KS*s is to change the agent's behaviour from reactive to cooperative. Cooperative agents can perform more efficiently at clearing all the tiles in the TileWorld. The agents can share their viewing range with other agents so that together they can get a bigger view of the TileWorld. They can also inform other agents of their target tile so that no two agents will move towards the same tile. This way, the agents can have better forward planning and can clear the tiles in the TileWorld efficiently. However, care must be taken to make sure that the planning time does not take too long as the TileWorld may change. If the TileWorld changes faster than the time taken for the agents to plan, then the agents will end up doing nothing as they will spend all their time planning. The current TileWorld-DARBS implementation does not have dynamic objects and as such, the TileWorld will not change, thus, giving the agents plenty of time to do their planning. However, future work on TileWorld-DARBS can introduce dynamic objects.

Dynamic objects can be easily implemented in the current TileWorld-DARBS by having another KS that randomly moves, creates, and deletes holes, obstacles and tiles. The rate and probability of change can be determined by user defined variables. Adding dynamic objects to TileWorld-DARBS does not require any changes to be made to the KSs except to the *Display TileWorld KS*. The only change required for the *Display TileWorld KS* is to add rules to check and update the GUI window for changes to the holes and obstacles in the TileWorld. This dynamic TileWorld adds more realism to the TileWorld and

enables further experiments to be carried out on the agents' behaviour in a dynamic environment.

A new experiment can also be carried out on TileWorld-DARBS to see how long it takes to clear up all the tiles in a static TileWorld. The experiments carried out so far showed the performance increase for each *Agent KS* as the *Agent KS*s are distributed across different number of processors. The new experiment will show the performance increase for solving a problem with different number of agents distributed across different processors. For example, the experiment can show how much faster ten *Agent KS*s on ten separate processors can clear up all the tiles in a static TileWorld compared to one *Agent KS*. This can also be used to show how efficient the agent rules are compared to other agent rules and can also be used to compare the characteristics of reactive agents with cooperative agents. Another future work experiment is to investigate the performance of the distributed blackboard system as the number of agents and APs increases after the BB's saturation point. This experiment would require a large number of *Agent KS*s and APs, probably around twenty to thirty *Agent KS*s and APs. The results of this experiment would help better understand the effects of the BB's saturation point as the number of agents and APs increases. Further improvements to the BB can then be tested with the same experiment to show the effects of the improvements on the BB's saturation point. One such BB improvement would be to parallelise access to the BB by allowing concurrent access to different partitions on the BB.

As for emDARBS, similar performance experiments could be carried out to show the effects of different hardware architectures and operating systems on the performance of a distributed blackboard system. Unfortunately, due to memory constraint issues on the

SARNet, the TileWorld test-bed could not be implemented in emDARBS. The next generation of SARNet, XaNet is currently being developed and has 128 megabytes of memory on each node. Future work would be to implement TileWorld-emDARBS on the XaNet and rerun the performance experiments. The results of the performance experiments on TileWorld-emDARBS are expected to be similar to the PC-based performance experiments. However, the actual results of the emDARBS performance experiments can help better characterise and guide future implementation of distributed blackboard systems in intelligent embedded systems.

As for the SARNUX operating system, future works include porting the operating system to the XaNet. This requires changing some of the low level function calls as the hardware layer has changed. However, the first thing that needs to be done is to reconfigure and rebuild the software development tool-chain. Only with a proper software development tool-chain can SARNUX be successfully ported to the XaNet. After porting SARNUX, work should be carried out to develop the advance debugging software that uses the JTAG. Advance debugging that includes runtime watches, program step through, and breakpoints are crucial in being able to develop complex software in an embedded system. This type of advance debugging can be accomplished by using In-Circuit Emulation (ICE) [135] or remote serial debugging with GNU GDB [136]. In-Circuit Emulation (ICE) requires some additional hardware support but offers more flexible debugging, whereas, remote serial debugging only requires a RS232 serial link but would require operating system support on the target (in this case, GDB support from the SARNUX).

Future works for emDARBS include reworking emDARBS to comply fully with the Embedded C++ working standard. This will make emDARBS a subset of DARBS that specially caters for embedded systems. This way, emDARBS application will be able to function in DARBS provided that there is no hardware dependent code and DARBS application can be ported to emDARBS by replacing non-Embedded C++ compliant codes with Embedded C++ compliant codes. The current emDARBS IPC layer that was created to mimic TCP/IP communication calls can be made more generalised to become a standard interface layer for Linux's TCP/IP applications to run on SARNUX. However, this is not ideal for processor intensive applications as the communication interface layer would add more overhead to the applications. A better way would be to change the applications' communication model to use SARNUX's CSP communication model.

Another future work for emDARBS is to add real-time features to the system. This is essential for real-time embedded systems. Examples of real-time features that can be added to emDARBS are information time-stamping and time deadline. The BB can add a time-stamp to the information that is added onto the BB to show when the information was added. This can help late KSs to pick up information that they missed or to pick the latest information and ignore the old information. The KSs can also add a time deadline to certain information that needs to be used before a certain deadline and to show when the information is no longer valid. This information can help other KSs to prioritise their tasks to process information that has the shortest time deadline first. The BB can also benefit from the time deadline information as the BB can perform periodic clean up on expired information to conserve memory capacity.

Another area that can be improved on is to introduce raw binary data on the BB. At the moment, only ASCII [137] string data can be stored on the BB. There can be occasions when large amounts of raw data need to be stored on the BB and converting all the raw data to ASCII string format can be time-consuming and will result in an enormous amount of data. This is impractical and a better way to store this information would be either to store the data on the BB in raw binary format or to compress the data and store it on the BB. If the data is really large, then a better solution would be to store the data in a separate location and only store the pointer to the data on the BB.

All these future works will bring emDARBS closer to the possibility of being implemented in future intelligent embedded systems. An example of how emDARBS may be implemented in future intelligent embedded systems is as shown in Figure 57.

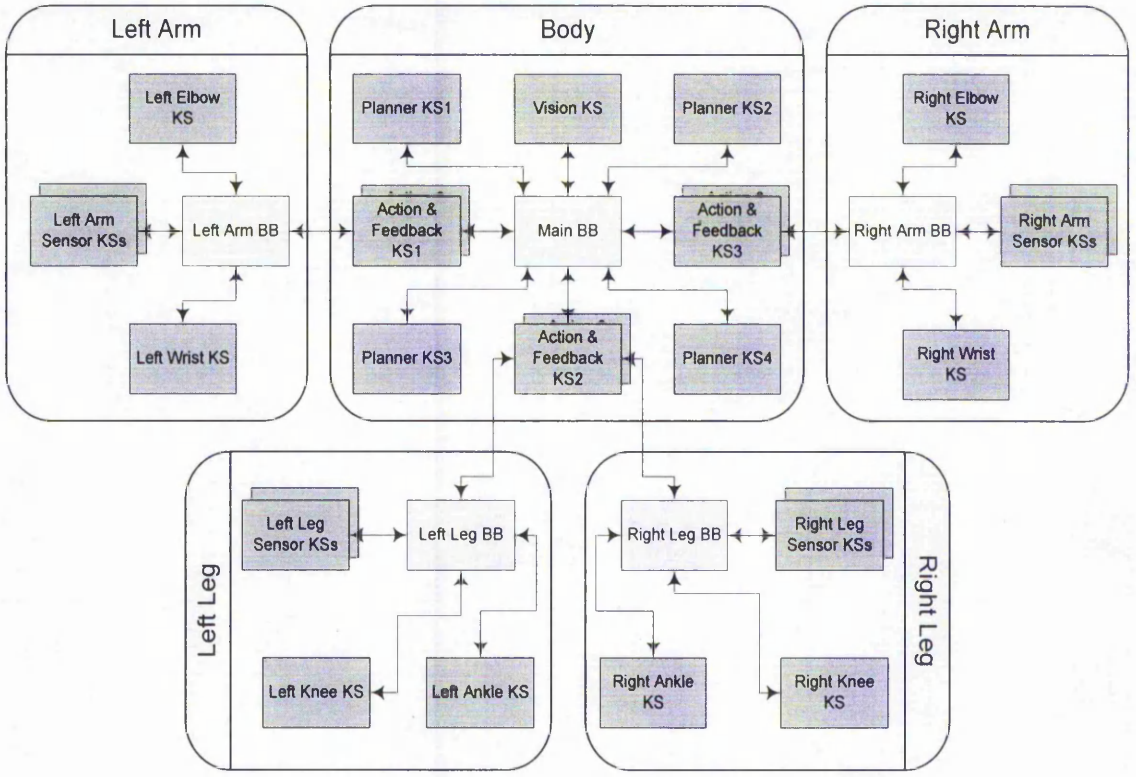


Figure 57. Example of distributed blackboard systems implementation in a robot

The example in Figure 57 shows how distributed blackboard systems can be implemented on an intelligent embedded system such as a robot. There are five BBs in this example; one for each limb of the robot and one more for the main body. The *Main BB* will store information regarding the overall goals of the robot and the top level status of the robot and its surroundings. The *Vision KS* will store high level visual information of the environment onto the *Main BB*. The *Planner KS*s will make plans on how to achieve the overall goals of the robot and store that information on the *Main BB*. The corresponding *Action & Feedback KS*s will then pick up the relevant plans and create an actions list and store that onto the corresponding limb BB (e.g. *Left Arm BB*, *Right Leg BB*, etc.). The corresponding KSs in that limb will control their respective motors to move the limb according to the actions list on the limb BB. The *Sensor KS*s in each limb will store sensor readings of the environment onto the limb BB and the corresponding *Action & Feedback KS*s will process the sensor readings and store the high level environment status onto the *Main BB*. The limb KSs (e.g. *Right Knee KS*, *Left Wrist KS*, etc.) will also have some basic reactions that do not need to go through the *Planner KS*s. For example, if the left wrist's temperature sensor detects high temperature from one direction, then the *Left Wrist KS* and possibly the whole arm KSs (depending on the amount of heat detected) will automatically react to that by quickly moving the wrist or the whole arm away from the source of the high temperature. This type of basic reaction will ensure that the robot can react in real-time. Although this robot example is still far from being a reality, the work carried out in this research has made a small step in realising the actual implementation of this robot.

References

[1] Callaghan, V., Colley, M., Clarke, G., Hagras, H., "The Cognitive Disappearance of the Computer: Intelligent Artifacts and Embedded Agents", *Summary paper for i3 2001 workshop WS4 on Cognitive Versus Physical Disappearance*, Porto, Portugal, 2001.

[2] Picton, P., "Introduction to neural networks", *England: Macmillan*, 1994.

[3] Negnevitsky, M., "Artificial Intelligence: A guide to intelligent systems", *UK: Addison-Wesley*, 2002.

[4] Koza, J. R., "Genetic Programming", *Cambridge: The MIT Press*, 1992.

[5] Hopgood, A. A., "Intelligent Systems for Engineers and Scientists", *CRC Press*, 2002.

[6] Brown, M. D., Fisher, R. B., "A Distributed Blackboard System for Vision Application", *BMVC90 Proceedings of the British Machine Vision Conference*, Oxford, UK, 1990, pp. 163-168.

[7] Canny, J. F., "A Computational Approach to Edge Detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol 8, No. 6, Nov 1986, pp. 679-698.

[8] Sobczak, R. S., Matthews, M. M., "A Massively Parallel Expert System Architecture for Chemical Structure Analysis", *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990, pp. 11-17.

[9] Sohi, G. S., Breach, S. E., Vijaykumar, T. N., "Multiscalar processors", *In 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 414-425.

[10] Lim, A. W., Lam, M. S., "Maximizing parallelism and minimizing synchronization with affine transforms", *In Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, ACM Press, 1997, pp. 201-214.

[11] Uhr, L., "Multi-computer architectures for artificial intelligence: Towards fast, robust, parallel systems", *Canada: John Wiley & Sons*, 1987.

[12] Occello, M., Demazeau, Y., "Building real time agents using parallel blackboards and its use for mobile robotics", *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics*, Volume: 2, 2-5 Oct 1994, pp. 1610-1615.

[13] Barr, M., "Programming Embedded Systems in C and C++", *O'Reilly*, 1999.

[14] Buzzard, G., Jacobson, D., Mackey, M., Marovitch, S., Wilkes, J., "An implementation of the Hamlyn sender-managed interface architecture", *In 2nd USENIX Symposium on Operating System Design and Implementation (OSDI)*, October 1996, pp. 245-260.

[15] Dragoni, A. F, Giorgini, P., "Self-monitoring distributed monitoring systems for nuclear power plants", In J. Mira, A. P. Del Pobl & M. Ali, Eds. *Methodology and Tools in Knowledge-Based Systems, Vol 1, Lecture Notes in Computer Science, No. 1415*, Berlin: Springer-Verlag, 1998.

[16] Stankovic, J.A., "Distributed Computing", *Readings in Distributed Computing Systems*, Casavant, T.L., Singhal, M., Eds., IEEE Computer Society Press, Los Alamitos, CA., 1994, pp. 6-30.

[17] Silva, J. L. da Jr., Shamberger, J., Ammer, M. J., Guo, C., Li, S., Shah, R., Tuan, T., Sheets, M., Rabaey, J. M., Nikolic, B., Sangiovanni-Vincentelli, A. L., Wright, P., "Design methodology for PicoRadio networks", *In Proceedings of the Design Automation and Test in Europe (DATE 2001)*, Munich, Germany, March 2001, pp. 314-325.

[18] Freund, U., Schorcht, G., Zerbe, V., "Hierarchical Simulation Techniques for the Design of Heterogeneous, Embedded Systems", *IFIP Workshop on Modelling of Microsystems*, Stirling, Scotland, 3-4 July 1997.

[19] Weicker, R. P., "Dhrystone: a synthetic systems programming benchmark", *Communications of the ACM*, Vol. 27, Issue 10, October 1984, pp. 1013-1030.

[20] "Digital Semiconductor SA-110 Microprocessor – Technical Reference Manual", *Digital Equipment Corporation*, Maynard, Massachusetts, October 1996.

[21] Akhmetshina, E., Gburzynskiand, P., Vizeacoumar, F., "PicOS: A Tiny Operating System for Extremely Small Embedded Platforms", *Proceedings of Embedded Systems and Applications (ESA'03)*, CSREA Press, 2003, pp. 116-122.

[22] Edwards, S., Lavagno, L., Lee, E. A., Sangiovanni-Vincentelli, A., "Design of Embedded Systems: Formal Models, Validation, and Synthesis", *Proceedings of the IEEE*, vol. 85, no. 3, March 1997, pp. 366-390.

[23] Yeh, Y.C., "Triple-Triple Redundant 777 Primary Flight Computer", *Proceedings of the 1996 IEEE Aerospace Applications Conference*, Vol. 1, 1996, pp. 293–307.

[24] Simunic, T., Benini, L., Glynn, P. W., Micheli, G. D., "Dynamic power management for portable systems", *Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM'00)*, August 2000, pp. 11-19.

[25] Fornaciari, W., Gubian, P., Sciuto, D., Silvano, C., "Power estimation of embedded systems: A hardware/software codesign approach", *IEEE Transactions on VLSI Systems*, Vol. 6, No. 2, June 1998, pp. 266-275.

[26] Shukla, S. K., Gupta, R. K., "A Model Checking Approach to Evaluating System Level Dynamic Power Management Policies for Embedded Systems", *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT'01)*, 2001, pp. 53-57.

[27] Li, D., Chou, P. H., Bagherzadeh, N., "Mode Selection and Mode-Dependency Modeling for Power-Aware Embedded Systems", *Proceedings of VLSI Design 2002*, pp. 697-704.

[28] Jerraya, A.A., Baghdadi, A., Cesario, W., Gauthier, L., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., INVITED PAPER, "Application-Specific Multiprocessor Systems-on-Chip", *Proceedings of The Tenth Workshop on Synthesis And System Integration of Mixed Technologies (SASIMI) 2001*, Nara, Japan, October 18-19 2001, pp. 317-324.

[29] Abdelzaher, T., Atkins, E., Shin, K., "QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control", *IEEE Transactions on Computers*, Vol. 49, No. 11, 2000, pp. 1170-1183.

[30] Kao, B., Garcia-Molina, H., "An Overview of Real-Time Database Systems", In W.A. Halang & A.D. Stoyenko, Eds., *Real-Time Computing*, NATO ASI Series F, 127, Springer-Verlag, 1994, pp. 261-282.

[31] Howard, A., Padgett, C., Brown, K., "Real Time Intelligent Target Detection and Analysis with Machine Vision", *Third International Symposium on Intelligent Automation and Control*, World Automation Congress, 11-16 June 2000.

[32] Washington, R., Golden, K., Bresina, J., Smith, D. E., Anderson, C., Smith, T., "Autonomous Rovers for Mars Exploration", *Proceedings of the IEEE Aerospace Conference*, IEEE, 1999.

[33] Negnevitsky, M., "Artificial Intelligence, A Guide to Intelligent Systems", *Addison Wesley*, 2002.

[34] Lesley Brown, ed. "The New Shorter Oxford English Dictionary on Historical Principles", Vol. 1, *Clarendon Press*, 1993, pp. 1387.

[35] Hsu, F.H., "Behind Deep Blue: Building the Computer That Defeated the World Chess Champion", *Princeton University Press*, 2002.

[36] Schaeffer, J., "The Games Computers (and People) Play", *Academic Press*, Vol. 50, Zelkowitz M.V., Eds., 2000, pp 189-266.

[37] Shortliffe, E.H., "MYCIN: a knowledge-based computer program applied to infectious diseases", *The First Annual Symposium on Computer Application in Medical Care*, 1977, pp. 66-69.

[38] Hopgood, A.A., "Artificial intelligence: hype or reality?", *IEEE Computer*, Vol. 6, May 2003, pp. 24-28. ISSN: 0018-9162.

[39] Lucey, C., "Is It Worth It? -Robot in Disguise- Trilobite vacuum cleaner", *Financial Times*, London, England, 29th November 2003, pp. 42.

[40] Freisleben, B., Kunkelmann, T., "Combining Fuzzy Logic and Neural Networks to Control an Autonomous Vehicle", *Proceedings IEEE International Conference on Fuzzy Systems*, 1993, pp. 321-326.

[41] Burton, A. R., Vladimirova, T., "A genetic algorithm for utilising neural network fitness evaluation for musical composition", *Proceedings of the 1997 International Conference on Artificial Neural Networks and Genetic Algorithms*, 1997, pp. 220-224.

[42] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., Mankovski, S., "High Level Multi-Agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example", *Third International Conference and Exhibition on the Practical Applications of Intelligent Agents and Multi-Agents (PAAM'98)*, 23-27 March 1998, London, UK.

[43] Oliveira, E., "Cooperative multi-agent system for an assembly robotics cell", *Robotics and Computer Integrated Manufacturing*, Vol. 11, No. 4, 1994, pp. 311-317.

[44] Huhns, M. N., Singh, M. P., eds., "Readings in Agents", *USA: Morgan Kaufmann Publisher*, 1998.

[45] Ferber, J., "Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence", *England: Addison Wesley*, 1999.

[46] Engelmore, R., Morgan, T., ed. "Blackboard Systems". *Great Britain: Addison Wesley*, 1988.

[47] Liscano, R., Fayek, R.E., Karam, G.M., "A Blackboard, Activity-Based Control Architecture for a Mobile Platform", *Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, July 7-10 1992, pp. 333-338.

[48] Sudha, R., Venkataraman, R., "A Blackboard-Based Cooperative System for Schema Integration", *IEEE Inteligent Systems & Their Applications*, Vol. 10, No. 3, June 1995, pp. 56-62.

[49] Montakhab, M. R., Adams, R. N., "Intelligent system for fault diagnosis on low voltage distribution networks", *IEE Proceedings - Generation, Transmission and Distribution*, Vol. 145, No. 5, September 1998, pp. 592-596.

[50] Jones, N. B., Spurgeon, S. K., Pont, M. J., Twiddle, J. A., Lim, C. L., Parikh, C. R., Goh, K. B., "Aspects of diagnostic schemes for biomedical and engineering systems", *IEE Proceedings - Science, Measurement and Technology*, Vol. 147, No. 6, November 2000, pp. 357-362.

[51] Hopgood, A. A., Woodcock, N., Hallam, N. J., Picton, P. D., "Interpreting Ultrasonic Images Using Rules, Algorithms and Neural Networks", *European Journal of NDT*, Vol. 2, No. 4, April 1993, pp. 135-149.

[52] Hopgood, A. A., "Rule-based control of a telecommunications network using the blackboard model", *Artificial Intelligence in Engineering*, Vol. 9, 1994, pp. 29-38.

[53] Hopgood, A.A., Phillips, H.J., Picton, P.D., Braithwaite, N.St.J., "Fuzzy logic in a blackboard system for controlling plasma deposition processes", *Artificial Intelligence in Engineering*, Vol. 12, 1998, pp. 253-260.

[54] Vranes, S., Stanojevic, M., "Integrating Multiple Paradigms within the Blackboard Framework", *IEEE Transactions on Software Engineering*, Vol. 21, No. 3, March 1995, pp. 244-262.

[55] Vranes, S., Stanojevic, M., Stevanovic, V., "BEST-based expert diagnostic system for the aluminum industry", *Computers in Industry*, Vol. 32, No. 1, Dec 5 1996, pp. 53-68.

[56] Vranes, S., Stanojevic, M., Stevanovic, V., Lucin, M., "INVEX: investment advisory expert system", *Expert Systems*, Vol. 13, No. 2, May 1996, pp. 105-119.

[57] Fathi, M., Holte, K., Lueg, C., Scharnetzki, R., "Knowledge-based Supervisory Process Control: Applying Fuzzy Sets to Blackboard Control Architecture", *Proceedings of the 2nd New Zealand Two-Stream International Conference on Artificial Neural Networks and Expert Systems (ANNES '95)*, 1995, pp. 153-156.

[58] McManus, J. W., "A Concurrent Distributed System for Aircraft Tactical Decision Generation", *IEEE/AIAA/NASA 9th Digital Avionics Systems Conference Proceedings*, New York, USA, IEEE, 1990, pp. 505-512.

[59] Nolle, L., Wong, K. C. P., Hopgood, A. A., "DARBS: A Distributed Blackboard System", *Research and Development in Intelligent Systems XVIII*, Bramer, Coenen and Preece (eds.), Springer, 2001, pp 161–170.

[60] Knowledge Technologies International, 2000. "GBB Products ~ Products ~ NetGBB" [online], *KTIWorld*, Available at: <http://www.ktiworld.com/GBB/products_netgbb.html> [Accessed on 12th July 2004]

[61] Naaman, M., Zaks, A., "Fractal Blackboard Framework", *Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering*, 1997, pp. 23-29.

[62] Lander, S. E., Staley, S. M., Corkill, D. D., "Designing Integrated Engineering Environments: Blackboard-Based Integration of Design and Analysis Tools", *Concurrent Engineering: Research and Application*, Vol. 4, No. 1, March 1996, pp.59-72.

[63] Lee, D. D., Seung, H. S., "Learning in intelligent embedded systems", *Proceedings of the Embedded Systems Workshop*, Cambridge, Massachusetts, USA, 29-31 March 1999, pp. 81-90.

[64] Hornby, G.S., Takamura, S., Yokono, J., Hanagata, O., Yamamoto, T., Fujita, M., "Evolving robust gaits with AIBO", *IEEE International Conference on Robotics and Automation*, 2000, pp. 3040-3045.

[65] Holve, R., Protzel, P., "Generating Fuzzy Rules for the Acceleration Control of an Adaptive Cruise Control System", *Proceedings of the Biennial Conference of the North American Fuzzy Information Processing Society - NAFIPS*, June 19th-22nd, 1996, pp. 451-455.

[66] Musliner, D. J., Hendler, J. A., Agrawala, A. K., Durfee, E. H., Strosnider, J. K., Paul, C. J., "The Challenges of Real-Time AI", *IEEE Computer*, Vol. 28, No. 1, January 1995.

[67] Ljungberg, M., Lucas, A., "The OASIS Air Traffic Management System", *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence*, PRICAI '92, 1992.

[68] Atkins, E. M., Durfee, E. H., Shin, K. G., "Autonomous Flight with CIRCA-II", *Autonomous Agents-99 Workshop on Autonomy Control Software*, May 1999.

[69] Abdelzaher, T. F., Atkins, E. M., Shin, K. G., "QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control", *IEEE Transactions on Computers*, Vol. 49, No. 11, November 2000, pp. 1170-1183.

[70] Hockney, R. W., Jeshope, C. R., "Parallel Computers 2: Architecture, Programming and Algorithms", 2nd Ed., *UK: IOP Publishing Ltd.*, 1988, pp. 1-53.

[71] Xavier, C., Iyengar, S. S., "Introduction to Parallel Algorithms", *USA: John Wiley & Sons*, 1998, pp. 1-29.

[72] Feitelson, D. G., Nitzberg, B., "Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860", *In Job Scheduling Strategies for Parallel Processing*, 1995, pp. 337-360.

[73] nCUBE Corporation, 2004. "nCUBE", [online], *nCUBE Corporation*, Available at: < http://www.ncube.com/index.html> [Accessed on 12th July 2004]

[74] Chiola, G., Ferscha, A., "Distributed Simulation of Timed Petri Nets: Exploiting the Net Structure to Obtain Efficiency", *Proceedings of the 14th International Conference on Application and Theory of Petri Nets 1993*, Chicago, June 1993, pp. 146-165.

[75] Intel Corp, 2004. "Intel Museum – Archives and Collections", *Intel Corporation*, Available at: <ftp://download.intel.com/intel/intelis/museum/research/arc_collect/timeline/TimelineChron.pdf>, can also be obtained from link at: <http://www.intel.com/intel/intelis/museum/archives/timeline/index.htm> [Accessed on 12th July 2004]

[76] Lewis, T. G., El-Rewini, H., Kim, I. K., "Introduction to parallel computing", *USA: Prentice Hall*, 1992, pp. 73-90.

[77] Chung, Y., Prasanna, V. K., Wang, C.-L., "A fast asynchronous algorithm for linear feature extraction on IBM SP-2", *In Proc. of IEEE Workshop on Computer Architecture for Machine Perception*, Como, Italy, Sept 1995, pp. 294-301.

[78] Funke, R., Lüling, R., Monien, B., Lücking, F., Blanke-Bohne, H., "An Optimized Reconfigurable Architecture for Transputer Networks", *Proc. of the 25th Hawaii Int. Conference on System Sciences (HICSS'92)*, Vol. 1, 1992, pp. 237-245.

[79] "ICR C416 - 16-port Dynamic Routing Switch for Transputer Links - Data Sheet", *IC-Routing Ltd.*, Nottingham, UK, 1996. Obtainable at: <http://eee.ntu.ac.uk/research/parallel/icrltd.html>

[80] O'Neill, B. C., Wong, K. L., Coulson, G. C., Hotchkiss, R., Ng, J. H., Clark, S., Thomas, P. D., Cawley, A., "A Distributed Parallel Processing System for the StrongARM Microprocessor". *Concurrent Systems Engineering*, Vol. 52, April 1998, pp 39-48.

[81] Hotchkiss, R., Wong, K. L., O'Neill, B. C., Coulson, G. C., Clark, S., Thomas, P. D., "The Building Blocks for a Parallel Network Incorporating the StrongARM Microprocessor", *The 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Las Vegas, Nevada, USA, July 1998, pp 1863-1870.

[82] Gasser, L., Branganza, C., Herman, N., "Implementing Distributed AI Systems Using MACE", *Proceedings of the 3rd Conference on Artificial Intelligence Applications*, IEEE Computer Soc. Press, 1987, pp. 315-320.

[83] Hu, X., Sun, W., "A unit cell-based design support system for composite structures", *Composite Structures*, Vol. 55, No. 3, Feb-March 2002, pp. 261-268.

[84] Brzykcy, G., Martinek, J., Meissner, A., Skrzypczynski, P., "Multi-agent blackboard architecture for a mobile robot", *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the Next Millennium IEEE.* Part Vol. 4, 2001, pp. 2369-2374.

[85] Pomeroy, B., Irving, R., "A Blackboard Approach for Diagnosis in Pilot's Associate", *IEEE Intelligent Systems & their Applications*, Vol. 5, No. 4, IEEE, August 1990. pp. 39-46.

[86] Corkill, D. D., "Collaborating Software: Blackboard and Multi-Agent Systems & the Future", *Proceedings of the International Lisp Conference*, New York, October 2003.

[87] Pollack, M. E., Ringuette, M., "Introducing the Tileworld: Experimentally Evaluating Agent Architectures", *Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI Press*, 1990, pp. 183-189.

[88] Ephrati, E., Pollack, M., Ur, S., "Deriving multi-agent coordination through filtering strategies", *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1995, pp. 679-685.

[89] Lees, M., Logan, B., Theodoropoulos, G., "Adaptive Optimistic Synchronisation For Muti-Agent Distributed Simulation", *Proceedings 17th European Simulation Multiconference*, June 2003, pp. 77-82.

[90] Kinny, D., Georgeff, M., Hendler, J., "Experiments in Optimal Sensing for Situated Agents", *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence (PRICAI'92)*, 1992, pp. 1176-1182.

[91] Uhrmacher, A.M., Schattenberg, B., "Agents in Discrete Event Simulation", In: Bargiela, A., Kerckhoffs, E., eds., *Proceedings of the 10TH European Simulation Symposium "Simulation in Industry - Simulation Technology: Science and Art" (ESS'98), SCS Publications*, 1998, pp. 129-136.

[92] Stevens, P., Pooley, R., "Using UML: Software engineering with objects and components", Updated Edition, *Addison-Wesley*, 2000.

[93] Matthew, N., Stones, R., "Beginning Linux Programming", UK: *Wrox Press*, 1997.

[94] Josuttis, N. M., "The C++ Standard Library: A Tutorial and Reference", UK: *Addison Wesley*, 1999.

[95] Dalheimer, M. K., "Programming with Qt", Germany: *O'Reilly*, 2002.

[96] Griffith, A., "KDE/QT Programming Bible", *John Wiley & Sons Inc.*, 2001.

[97] Hughes, C., Hughes, T., "Object-Oriented Multithreading Using C++", *Wiley*, 1997.

[98] Butenhof, D. R., "Programming with POSIX Threads", *Addison Wesley*, July 1997.

[99] Lapin, L. L., "Statistics for modern business decisions", 3rd Edition, USA:*Harcourt Brace Jovanovich*, 1981.

[100] Harper, W. M., "Statistics", 6th Edition, UK:*Financial Times Prentice Hall*, 1991.

[101] Tanenbaum, A. S., "A comparison of three microkernels", *Journal of Supercomputing*, Vol. 9, No. 1/2, 1995, pp. 7-22.

[102] Humphrey, M., Wallace, G., Stankovic, J., "Kernel-Level Threads for Dynamic, Hard Real-Time Environment", *16th IEEE Real Time Systems Symposium*, 1995, pp. 38-48.

[103] Bovet, D. P., Cesati, M., "Understanding the Linux Kernel", 2nd Edition, USA: *O'Reilly*, 2002.

[104] Dignum, F., Morley, D., Sonenberg, E. A., Cavedon, L., "Towards socially sophisticated BDI agents", *In Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS'2000)*, 2000, pp. 111-118.

[105] Schroeder, M., M˜ora, I. A., Pereira, L. M., "A Deliberative and Reactive Diagnosis Agent Based on Logic Programming", In Muller, J. P., Wooldridge, M., Jennings, N. R., eds., Intelligent Agents III, *Springer-Verlag*, 1996, pp. 293-307.

[106] Krogh, C., "The rights of agents", In Wooldridge, M., Muller, J. P., Tambe, M., eds., Intelligent Agents Volume II --- Proceedings of the IJCAI-95 Workshop on Agent Theories, Architectures, and Languages, *Springer-Verlag*, 1996, pp. 1-16.

[107] Li, G., Hopgood, A. A., Weller, M. J., "Shifting Matrix Management: a model for multi-agent cooperation", *Engineering Applications of Artificial Intelligence*, Vol. 16, 2003, pp. 191-201. ISSN: 0952-1976.

[108] Sun, X.-H., Zhu, J., "Performance Considerations of Shared Virtual Memory Machines", *IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No.11*, 1995, pp. 1185-1194.

[109] Na'im, O., Teruel, A., "ANDES: A Performance Analyzer for Parallel Programs", *In Transputer and Occam Research: New Directions, 16th Technical Meeting of the World Occam and Transputer User Group (WoTUG-16)*, Sheffield, UK, Vol. 33, March 1993, pp. 91-99.

[110] Behr, P. M., Pletner, S., Sodan, A. C., "PowerMANNA: A parallel architecture based on the powerPC MPC620", *IEEE High-Performance Computer Architecture Symposium Proceedings*, 2000, pp. 277-286.

[111] Jones, A., Hopper, A., "Handling Audio and Video Streams in a Distributed Environment", *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993, pp. 231-243.

[112] Manners, D., "END OF THE LINE", *Electronics Weekly*, Inside View, 16th July 1997, pp. 14.

[113] Wong, K. L., "A Message Controller for Distributed Processing Systems", *Ph.D. thesis*, Nottingham Trent University, UK, 2000.

[114] Liew, E. W. K., Kaye, D., O'Neill, B. C., Clark, S., "Operating System Support for StrongARM Multi-Processor Communications", *Proceedings of the ISCA 13th International Conference on Parallel and Distributed Computing Systems*, August 2000, pp. 334-339.

[115] Walters, E. G., "The Essential Guide to Computing: The Story of Information Technology", *Prentice Hall*, 2000, pp. 128-129.

[116] O'Neill, B.C., Moore, P.W., Clark, S., "A Single Chip Solution for Distributed Processing Systems", *Proceedings of Communicating Process Architectures 2003, IOS Press*, 7-10 September 2003, pp 83-90.

[117] Hotchkiss, R., O'Neill, B. C., Clark, S., "Fault Tolerance for an Embedded Wormhole Switched Network", *Parelec'2000, IEEE Computer Society proceedings*, August 2000, pp. 79-83.

[118] Ng, J. H., O'Neill, B. C., Clark, S., "A PC Interface Board for Parallel ARM Processor Network", *PREP 2000, IEE*, April 2000, pp. 469-474.

[119] Hunger, S., "Debian GNU/Linux Bible", *John Wiley & Sons*, 2001.

[120] Stallman, R. M., Gay, J., Lessig, L., "Free Software, Free Society: Selected Essays of Richard M. Stallman", *Free Software Foundation*, 2002.

[121] Liew, E. W. K., O'Neill, B. C., Clark, S., "Porting Transputer Application to Multi-Processors StrongARM system", *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM'2001) under the International Parallel & Distributed Processing Symposium (IPDPS 2001)*, San Francisco, USA, April 23rd, 2001, pp. 30113b.

[122] Hoare, C. A. R., "Communicating Sequential Processes", *Prentice Hall*, 1985.

[123] Glover, I. A., Grant, P. M., "Digital Communications", UK: *Prentice Hall*, 1998, pp. 658-682.

[124] Liew, E. W. K., "A Distributed Real-Time Operating System for a Multi-Processor StrongARM Network", *Ph.D. Thesis*, Nottingham Trent University, February 2002.

[125] Raynal, M., "Algorithms for Mutual Exclusion", UK: *North Oxford Academic*, 1986.

[126] Schildt, H., "The Complete Reference: C++", 3rd Ed., USA: *McGraw Hill*, 1998, pp. 807-927.

[127] Stallman, R., Pesch, R., Shebs, S., "Debugging With Gdb: The Gnu Source-Level Debugger", *Free Software Foundation*, 2002.

[128] Kowalski, R., Sadri, F., "Towards a unified agent architecture that combines rationality with reactivity", In Pedreschi, D., Zaniolo, C., editors, *Proceedings of LID-96*, Vol. 1154 of LNAI, 1996, pp. 137-149.

[129] Sichman, J. S., Demazeau, Y., Boissier, O., "When Can Knowledge-Based Systems be Called Agents", *In Proceedings of IX Brazilian Symposium on Artificial Intelligence*, Rio de Janeiro, Brazil, 1992, pp. 172-185.

[130] Massa, A., "Embedded Software Development with ECos", *Prentice Hall*, 2002.

[131] Loukides, M., Oram, A., "Programming with GNU Software", UK: *O'Reilly*, 1996.

[132] Institute of Electrical and Electronic Engineers Standards, IEEE 1149.1-2001, IEEE Standard Test Access Port and Boundary Scan Architecture, 2001.

[133] Plauger, P.J., "Embedded C++", *C/C++ Users Journal*, February 1997, p. 35.

[134] Plauger, P.J., "Embedded C++: An Overview", *Embedded Systems Programming*, December 1997.

[135] Babb, J., Tessier, R., Dahl, M., Hanono, S., Hoki, D., Agarwal, A., "Logic emulation with Virtual Wires", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 16, No. 6, June 1997, pp. 609-626.

[136] Stallman, R., Pesch, R., Shebs, S., "Debugging With Gdb: The Gnu Source-Level Debugger", *Free Software Foundation*, 2002.

[137] Mackenzie, C. E., "Coded character sets, history & development", *Addison Wesley Longman Publishing*, 1980. ISBN 0-201-14460-3.

Appendix A : List of publications

Choy, K. W., Hopgood, A. A., Nolle, L., O'Neill, B. C., "Performance of a multi-agent simulation on a distributed blackboard system", Accepted for publication in *International Journal of Simulation Systems, Science & Technology*, ISSN: 1473-8031 (Print) / 1473-804x (Online).

Choy, K. W., Hopgood, A. A., Nolle, L., O'Neill, B. C., "Implementing a blackboard system in a distributed processing network", *Expert Update*, Vol. 7, No.1, Spring 2004, pp. 16-24. ISSN: 1465-4091.

Choy, K. W., Hopgood, A. A., Nolle, L., O'Neill, B. C., "Implementation of a tileworld testbed on a distributed blackboard system", *Proceedings of ESM2004 - the 18th European Simulation Multiconference*, Magdeburg, Germany, June 2004, pp. 129-135. ISBN: 3-936150-35-4 (book) / ISBN: 3-936150-36-2 (CD)

Choy, K. W., Hopgood, A. A., Nolle, L., O'Neill, B. C., "Design and Implementation of an Inter-Process Communication Model for an Embedded Distributed Processing Network", *International Conference on Software Engineering Research and Practice (SERP'03)*, Las Vegas, June 2003, pp. 239-245.

Choy, K. W., Hopgood, A. A., Nolle, L., O'Neill, B. C., "Inter-Process Communication for SARNet", Abstract and presentation at *The Embedded Systems Show ESS2003*, London, UK, 14-16 May, 2003.

Choy, K. W., Hopgood, A. A., Nolle, L., O'Neill, B. C., "emDARBS Inter-Process Communication Protocol", Abstract and poster presented at *ES2002*, Cambridge, December 2002.

Appendix B : DARBS Command

```
////////////////////////////////////////////////////////
//              DARBS return commands
////////////////////////////////////////////////////////
// This commands are returned by the blackboard server.
// The return commands are self-explanatory.
//--------------------------------------------------------
//
#define PARTITION_NOT_FOUND      "partition not found!"
#define DARBS_NO_NAMES_DEFINED   "no partitions defined!"
#define DARBS_ERROR              "error"
#define DARBS_ERROR_INSUFFICIENT_ARG   "error, insufficient arguments"
#define DARBS_CONFIRM            "okay"
#define DARBS_COMMAND_NOT_FOUND  "command not found"
#define DARBS_FALSE              "false"
#define DARBS_TRUE               "true"
#define DARBS_NO_MATCH_FOUND     "no match found"
// This is a special marker on return string to give the caller
// additional information on the success/failure of the command
#define ADDITIONAL_INFO_STRING   "{*:*:EXTRA_INFO}"
// Gives info to the user of where the error occured in the
// command.
#define ERROR_AT                 "@"
//
//--------------------------------------------------------

////////////////////////////////////////////////////////
//              ARBS/DARBS commands
////////////////////////////////////////////////////////
// These are commands that are sent to the blackboard
// from the client.
// General info: Pattern and partitions can contain
// spaces as long as they are enclosed in [], or (),
// or { } brackets. The word brackets from here onwards
// refer to [], (), or { }. Brackets can be interpreted as
// a list of items separated by spaces or an item that
// contain spaces depending on the command being used.
//--------------------------------------------------------
//

#define DARBS_ADD                "add"
// To add a pattern to a particular partition. Using []
// would mean that the pattern or partition contain spaces.
// Usage: add <pattern> <partition>
// Return: On success - okay
//                   partition <partition> changed! (broadcasted)
//         On failure - error
```

```
#define DARBS_ADD_MULTI         "add_multi"
// To add multiple patterns to a particular partition. Using []
// would mean that the pattern or partition contain spaces. Each
// pattern is separated by a space
// Usage: add_multi <pattern1> <pattern2> ... <patternN> <partition>
// Return: On success - okay
//                 partition <partition> changed! (broadcasted)
//       On failure - error

#define DARBS_DEL               "del_first"
// To delete the first occurance of a pattern in a partition
// Usage: del_first <pattern> <partition>
// Return: On success - okay
//                 partition <partition> changed! (broadcasted)
//       On failure - error

#define DARBS_DEL_ALL           "del_all"
// To delete the all occurances of a pattern in a partition
// Usage: del_all <pattern> <partition>
// Return: On success - okay
//                 partition <partition> changed! (broadcasted)
//       On failure - error

#define DARBS_CLR               "clr_partition"
// To clear everything in a partition but not remove the partition
// Usage: clr_partition <partition>
// Return: On success - okay
//                 partition <partition> changed! (broadcasted)
//       On failure - error

#define DARBS_CLR_ALL           "clr_board"
// To clear everything on the blackboard (all partitions removed)
// Usage: clr_board
// Return: On success - okay
//                 partition changed! (broadcasted)
//       On failure - error

#define DARBS_RET               "ret_first"
// To return the first occurances of a pattern in a partition.
// Pattern are match with ? and must be exactly the same pattern
// as the data pattern on the blackboard. There can also be a ??
// query which return the rest of the string.
// eg. on partition part1
//    [This is data 1][This is data 2][This 1 data 2]
//    ret_first [This ? data 2] part1
//    -> true [ [ is ] ]
//    ret_first [This ?? ] part1
//    -> true [ [ is data 1 ] ]
//
// Usage: ret_first <pattern with ?> <partition>
```

```
// Return: On success - true [ <matching word for ?> ]
//         On failure - false

#define DARBS_RET_ALL           "ret_all"
// To return all the occurances of a pattern in a partition.
// Pattern are match with ? and must be exactly the same pattern
// as the data pattern on the blackboard. There can also be a ??
// query which return the rest of the string.
// eg. on partition part1
//     [This is data 1][This is data 2][This 1 data 2]
//     ret_all [This ? data 2] part1
//     -> true [ [ is ] [ 1 ] ]
//     ret_all [This ?? ] part1
//     -> true [ [ is data 1 ] [ is data 2 ] [ 1 data 2 ] ]
//
// Usage: ret_all <pattern with ?> <partition>
// Return: On success - true [ <matching word for ?> ]
//         On failure - false

#define DARBS_GET               "get_contents"
// To list all the contents of a partition
// Usage: get_contents <partition>
// Return: On success - <contents in partition>
//         On failure - error

#define DARBS_SETUP_BLACKBOARD  "setup_blackboard"
// To clear the blackboard and put a new partition or
// a list of new partitions onto the blackboard. The list
// of partitions can be stored in [], (), or {}.
// This does not permit partition name with spaces in
// between. However, 'add' command allows spaces in partition
// name by using the [], (), or {} brackets.
// Note: An ambiguity occurs to differentiate partition name
//       with spaces in them and without spaces on the broadcasted message
// Usage: setup_blackboard <partition or list of partitions>
// Return: On success - okay
//                      partition <partition or list of partitions> changed! (broadcasted)
//         On failure - error

#define DARBS_ADD_PARTITIONS    "add_partitions"
// To add a list of new partitions to the blackboard. The
// list of partitions must be enclosed in [], (), or {}.
// This does not permit partition name with spaces in
// between. However, 'add' command allows spaces in partition
// name by using the [], (), or {} brackets.
// Note: An ambiguity occurs to differentiate partition name
//       with spaces in them and without spaces on the broadcasted message
// Usage: add_partitions <partition or list of partitions>
// Return: On success - okay
//                      partition <partition or list of partitions> changed! (broadcasted)
```

```
//        On failure - error

#define DARBS_GET_PARTITIONS      "get_partitions"
// To list all the partitions in the blackboard
// Usage: get_partitions
// Return: On success - <list of partitions in blackboard>
//        On failure - error

#define DARBS_PARTITION_EXIST     "partition_exist"
// To check if a partition exist on the blackboard.
// Usage: partition_exist <partition>
// Return: On success - true (partition exist)
//        On failure - false (partition does not exist)

#define DARBS_PARTITION_NOT_EXIST "partition_not_exist"
// To check if a partition does not exist on the blackboard.
// Usage: partition_not_exist <partition>
// Return: On success - true (partition does not exist)
//        On failure - false (partition exist)

#define DARBS_ON_PARTITION        "on_partition"
// To check if a pattern is on a partition or not.
// Partition name can contain space as long as it is enclosed
// in brackets.
// Note: There is a bug on this pattern checking if the
//       patterns on the partition is not separated by brackets.
// Usage: on_partition <pattern> <partition>
// Return: On success - true (pattern exist on partition)
//        On failure - false (pattern or partition does not exist)

#define DARBS_NOT_ON_PARTITION    "not_on_partition"
// To check if a pattern does not exist on a partition or not.
// Partition name can contain space as long as it is enclosed
// in brackets.
// Note: There is a bug on this pattern checking if the
//       patterns on the partition is not separated by brackets.
// Usage: not_on_partition <pattern> <partition>
// Return: On success - true (pattern or partition does not exist)
//        On failure - false (pattern exist on partition)

#define DARBS_ON_BLACKBOARD       "on_blackboard"
// To check if a pattern is on the blackboard or not.
// It checks all the partitions in the blackboard for the
// pattern. Partition name can contain space as long as
// it is enclosed in brackets.
// Note: There is a bug on this pattern checking if the
//       patterns on the partition is not separated by brackets.
// Usage: on_blackboard <pattern>
// Return: On success - true (pattern exist on the blackboard)
//        On failure - false (pattern does not exist on the blackboard)
```

```
#define DARBS_NOT_ON_BLACKBOARD   "not_on_blackboard"
// To check if a pattern does not exist on the blackboard.
// It checks all the partitions in the blackboard for the
// pattern. Partition name can contain space as long as
// it is enclosed in brackets.
// Note: There is a bug on this pattern checking if the
//       patterns on the partition is not separated by brackets.
// Usage: not_on_blackboard <pattern>
// Return: On success - true (pattern does not exist on the blackboard)
//         On failure - false (pattern exist on the blackboard)

#define DARBS_REPLACE             "replace"
// To replace a pattern in a particular partition with another pattern.
// Using [] would mean that the pattern or partition contain spaces.
// Usage: replace <pattern to be replace> <in partition> <replace pattern>
// Return: On success - okay
//                      partition <partition> changed! (broadcasted)
//         On failure - no match found
//                        when the pattern to be replace cannot be found
//                      partition not found!
//                        when the partition is not found

#define DARBS_REPLACE_MULTI       "replace_multi"
// CHANGED COMMAND - This command would now only perform the replace when
// all the pattern is a match. Changed on 17/11/03, KW Choy. The
// ADDITIONAL_INFO_STRING marker is not used anymore. Instead the ERROR_AT
// marker is used on the return message of an error, to show where the
// error was.
//
// To do multiple replace in one go. Function the same as DARBS_REPLACE but
// can do multiple. A special ADDITIONAL_INFO_STRING marker is added to the
// end of an error message to indicate which partition was successfully
// changed. For example if pattern1 on partition1 was successfully replaced
// but pattern2 cannot be found on partition2 then the return error message
// would be :-
//          no match found {*:*:EXTRA_INFO} [partition1]
// If there is no ADDITIONAL_INFO_STRING marker on the error message meaning
// nothing was changed.
// Using [] would mean that the pattern or partition contain spaces.
// Usage: replace <pattern1 to be replace> <in partition1> <replace pattern1> \
//             <pattern2 to be replace> <in partition2> <replace pattern2> ...
// Return: On success - okay
//                      partition <partition1> changed! (broadcasted)
//                      partition <partition2> changed! (broadcasted) ...
//         On failure - no match found
//                        when at least one of the pattern to be
//                        replace cannot be found
//                      partition not found!
//                        when at least one of the partition is not found
```

Appendix C : TileWorld KSs' rules

Initiator KS

- Clear_Blackboard
  - Clear the entire blackboard
- Init_TileWorld
  - Set the size of the TileWorld
  - Set the number of agents in the TileWorld
  - Set the number of holes in the TileWorld
  - Set the number of obstacles in the TileWorld
  - Set the number of tiles in the TileWorld
- Create_TileWorld
  - Generate a random position within the size of the TileWorld for each of the agents, holes, obstacles, and tiles in the TileWorld.
  - Store this information on the Blackboard.

Display TileWorld KS

- Display_Initial_Screen
  - Draw the grids in the TileWorld and label it
- Update_Agent_Display
  - Find the location of all the agents in the TileWorld from the Blackboard and display them accordingly.
- Update_Hole_Display
  - Find the location of all the holes in the TileWorld from the Blackboard and display them accordingly.
- Update_Obstacle_Display
  - Find the location of all the obstacles in the TileWorld from the Blackboard and display them accordingly.
- Update_Tile_Display
  - Find the location of all the tiles in the TileWorld from the Blackboard and display them accordingly.
- Update_Total_Objects_Display
  - Find out the objects that are currently being display in the TileWorld.
- Update_Deleted_Tile
  - Delete the objects that are no longer on the Blackboard from the displayed TileWorld.

Agent KS

- Initialise_Agent
  - Setup the viewing range the agent can see
  - Initialise the internal state of mind of the agent
  - Start the agent in Generate SearchSpace State.

- Update_Internal_Status
  - Reset back the agent's internal state of mind to the start state. This is required when the agent is interrupted and restarts itself.
  - Start the agent in Generate SearchSpace State.
- Generate_SearchSpace_State
  - Find out the coordinates that the agent can see based on the viewing range parameter that has been setup.
  - Find out the last coordinates that can be seen.
  - Set the agent to Look At Environment state
- Look_At_Environment_State1
  - Look at the contents of the environment that the agent can view and place that information into the agent's view partition
- Look_At_Environment_State2
  - Change the agent state to Thinking state as soon as the last coordinates that can be view by the agent is viewed.
- Is_It_Exploring_State
  - Change the agent state to Exploring state if the agent is in Thinking state and the agent is currently carrying no tile and there is no tile within the viewing range OR
  - Change the agent state to Exploring state if the agent is in Thinking state and the agent is currently carrying a tile and there is no hole within the viewing range.
- Is_It_Moving_To_Tile_State
  - Change the agent state to Moving To Tile state if the agent is in Thinking state and the agent is not carrying a tile and there is a tile within the viewing range and the tile is not on the same grid as the agent.
- Is_It_Hole_Filling_State
  - Change the agent state to Hole Filling state if the agent is in Thinking state and the agent is carrying a tile and the agent is standing on a grid with a hole.
- Is_It_Moving_To_Hole_State
  - Change the agent state to Moving To Hole state if the agent is in Thinking state and the agent is carrying a tile and there is a hole within the viewing range and the hole is not on the same grid as the agent.
- Is_It_Picking_Up_Tile_State
  - Change the agent state to Picking Up Tile state if the agent is in Thinking state and the agent is not carrying a tile and the agent is standing on a grid with a tile.
- Generate_Possible_Moves
  - Generate the 4 or less possible moves that the agent can make and store those moves in the agent's possible moves partition provided that the agent is in Exploring state or Moving To Tile state or Moving To Hole state. Finally added Check North Move Validity flag to the possible moves partition.
- Is_North_Move_Valid
  - If Check North Move Validity flag is set and the north move is valid then set the result on the possible moves partition. Change the Check North Move Validity flag to Check East Move Validity flag.
- Is_North_Move_NotValid

- If Check North Move Validity flag is set and the north move is not valid then set the result on the possible moves partition. Change the Check North Move Validity flag to Check East Move Validity flag.

- Is_East_Move_Valid
  - If Check East Move Validity flag is set and the east move is valid then set the result on the possible moves partition. Change the Check East Move Validity flag to Check South Move Validity flag.
- Is_East_Move_NotValid
  - If Check East Move Validity flag is set and the east move is not valid then set the result on the possible moves partition. Change the Check East Move Validity flag to Check South Move Validity flag.
- Is_South_Move_Valid
  - If Check South Move Validity flag is set and the south move is valid then set the result on the possible moves partition. Change the Check South Move Validity flag to Check West Move Validity flag.
- Is_South_Move_NotValid
  - If Check South Move Validity flag is set and the south move is not valid then set the result on the possible moves partition. Change the Check South Move Validity flag to Check West Move Validity flag.
- Is_West_Move_Valid
  - If Check West Move Validity flag is set and the west move is valid then set the result on the possible moves partition. Change the Check West Move Validity flag to Finish Checking Moves flag.
- Is_West_Move_NotValid
  - If Check West Move Validity flag is set and the west move is not valid then set the result on the possible moves partition. Change the Check West Move Validity flag to Finish Checking Moves flag.
- Exploring_State
  - If agent is in Exploring state, generate a random step based on the possible moves and the last made move. Store the random generated step on the agent's state of mind partition. Change agent's Exploring state to Making A Move state.
- Moving_To_Tile_State
  - If agent is in Moving To Tile state, clear the current closest tile information from the tile calculation partition. Change the agent's state to Get Tile Distance state.
- Moving_To_Hole_State
  - If agent is in Moving To Hole state, clear the current closest hole information from the hole calculation partition. Change the agent's state to Get Hole Distance state.
- Get_Tile_Distance_State
  - If agent is in Get Tile Distance state, calculate the distance of all the tiles within the agent's viewing range and store that information onto the agent's tile calculation partition. Change the agent's state to Finding Closest Tile state.
- Get_Hole_Distance_State
  - If agent is in Get Hole Distance state, calculate the distance of all the holes within the agent's viewing range and store that information onto the

agent's hole calculation partition. Change the agent's state to Finding Closest Hole state.

- Find_Closest_Tile_State
  - If agent is in Finding Closest Tile state, pick the closest tile in the tile calculation partition and store that in the agent's state of mind partition. Change agent's state to Generate Step Closer To Tile state.
- Find_Closest_Hole_State
  - If agent is in Finding Closest Hole state, pick the closest hole in the hole calculation partition and store that in the agent's state of mind partition. Change agent's state to Generate Step Closer To Hole state.
- Generate_Step_Closer_To_Tile_State
  - If agent is in Generate Step Closer To Tile state, generate a random step closer to the closest tile based on the possible moves, last move made, and the coordinates of the closest tile. Store the random generated step on the agent's state of mind partition. Change the agent's state to Making A Move state.
- Generate_Step_Closer_To_Hole_State
  - If agent is in Generate Step Closer To Hole state, generate a random step closer to the closest hole based on the possible moves, last move made, and the coordinates of the closest hole. Store the random generated step on the agent's state of mind partition. Change the agent's state to Making A Move state.
- Pick_Up_Tile_State
  - If agent is in Picking Up Tile state, pick up the tile from the same grid and change state back to Generate Searchspace state.
- Hole_Filling_State
  - If agent is in Hole Filling state, drop the tile it is carrying into the hole it is standing on and add up the current score of the agent with the score it made from filling the hole with a tile. Change agent's state back to Generate Searchspace state.
- Making_Move_State
  - If agent is in Making A Move state, record the current position of the agent as the last move and move the agent to the new location as stored in the agent's state of mind. Change agent's state back to Generate Searchspace state.

Appendix D : Alternative trend line graphs

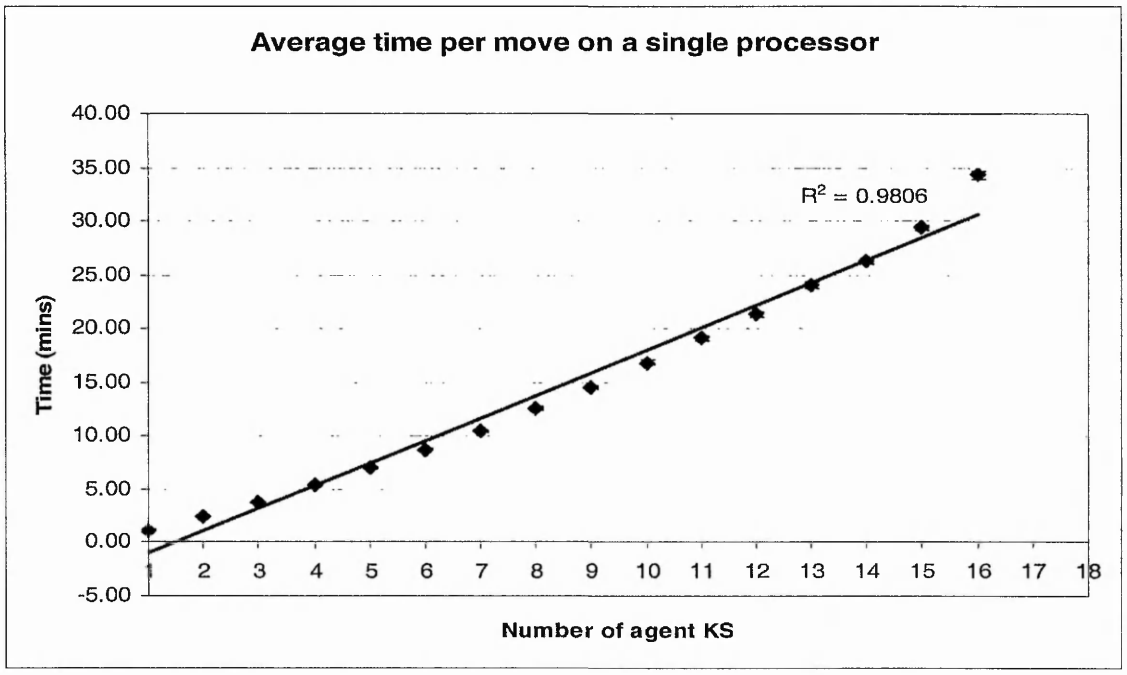


Figure 58. Average time per move on a single processor with linear function trend line

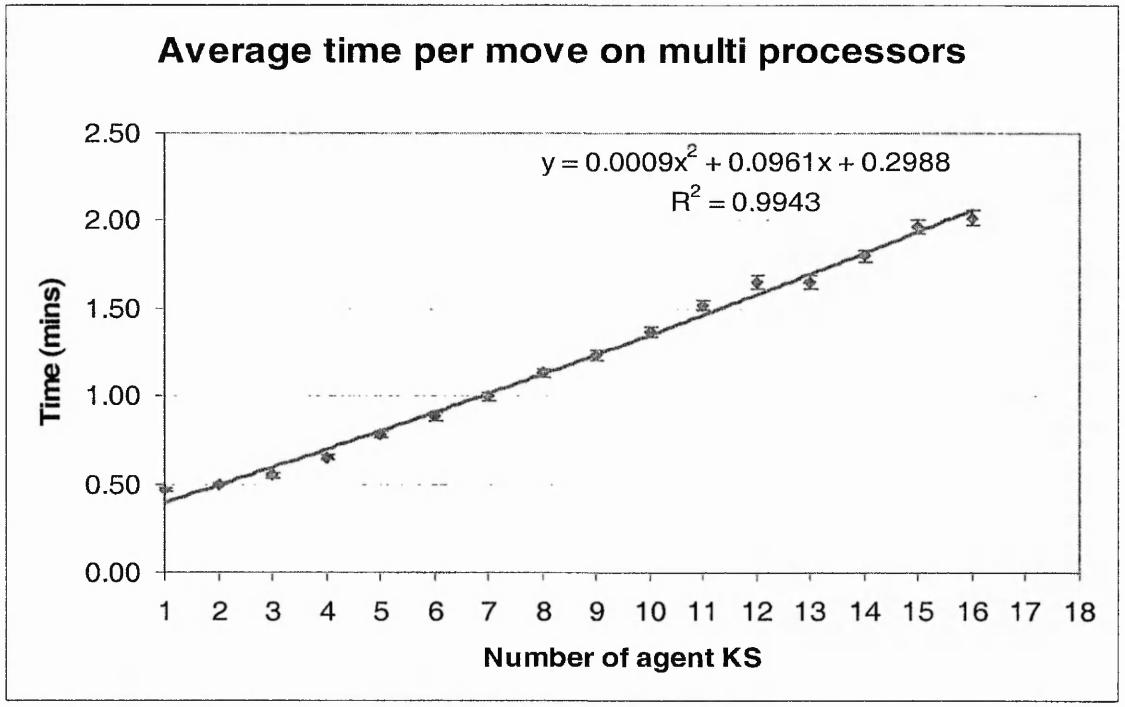


Figure 59. Average time per move on multi processors with polynomial function trend line

Appendix E : Table of results for comparing distributed and non-distributed performance

Singleprocessor (B07-12)							
Number of agents	Average time per move	Time in minutes	Standard deviation	Standard deviation (mins)	Normalise over 1 agent system (slow-down factor)	Standard error of mean (in mins) = standard deviation / sqrt(number of sample)	Ideal normalise values
1	00:01:11	1.18333	00:00:08	0.13333	1.0000	0.01886	1.00
2	00:02:28	2.46667	00:00:20	0.33333	2.0845	0.03333	2.00
3	00:03:49	3.81667	00:00:32	0.53333	3.2254	0.04355	3.00
4	00:05:30	5.50000	00:00:59	0.98333	4.6479	0.06953	4.00
5	00:07:04	7.06667	00:01:13	1.21667	5.9718	0.07695	5.00
6	00:08:46	8.76667	00:01:51	1.85000	7.4085	0.10681	6.00
7	00:10:26	10.43333	00:01:57	1.95000	8.8169	0.10423	7.00
8	00:12:33	12.55000	00:02:50	2.83333	10.6056	0.14167	8.00
9	00:14:30	14.50000	00:03:03	3.05000	12.2535	0.14378	9.00
10	00:16:49	16.81667	00:04:19	4.31667	14.2113	0.19305	10.00
11	00:19:05	19.08333	00:04:35	4.58333	16.1268	0.19543	11.00
12	00:21:19	21.31667	00:04:21	4.35000	18.0141	0.17759	12.00
13	00:24:03	24.05000	00:06:38	6.63333	20.3239	0.26018	13.00
14	00:26:15	26.25000	00:06:07	6.11667	22.1831	0.23119	14.00
15	00:29:22	29.36667	00:07:04	7.06667	24.8169	0.25804	15.00
16	00:34:17	34.28333	00:11:43	11.71667	28.9718	0.41425	16.00

Multiprocessors							
Number of agents	Average time per move	Time in minutes	Standard deviation	Standard deviation (mins)	Normalise over 1 agent system (slow-down factor)	Standard error of mean (in mins) = standard deviation / sqrt(number of sample)	Ideal normalise values
1	00:00:28	0.46667	00:00:04	0.06667	1.0000	0.00943	1.00
2	00:00:30	0.50000	00:00:05	0.08333	1.0714	0.00833	1.00
3	00:00:33	0.55000	00:00:10	0.16667	1.1786	0.01361	1.00
4	00:00:39	0.65000	00:00:12	0.20000	1.3929	0.01414	1.00
5	00:00:47	0.78333	00:00:18	0.30000	1.6786	0.01897	1.00
6	00:00:53	0.88333	00:00:23	0.38333	1.8929	0.02213	1.00
7	00:01:00	1.00000	00:00:25	0.41667	2.1429	0.02227	1.00
8	00:01:08	1.13333	00:00:31	0.51667	2.4286	0.02583	1.00
9	00:01:14	1.23333	00:00:35	0.58333	2.6429	0.02750	1.00
10	00:01:22	1.36667	00:00:41	0.68333	2.9286	0.03056	1.00
11	00:01:31	1.51667	00:00:46	0.76667	3.2500	0.03269	1.00
12	00:01:39	1.65000	00:00:51	0.85000	3.5357	0.03470	1.00
13	00:01:39	1.65000	00:00:56	0.93333	3.5357	0.03661	1.00
14	00:01:48	1.80000	00:00:53	0.88333	3.8571	0.03339	1.00
15	00:01:58	1.96667	00:01:05	1.08333	4.2143	0.03956	1.00
16	00:02:01	2.01667	00:01:12	1.20000	4.3214	0.04243	1.00

Appendix F : Table of results for speedup and efficiency of varying number of agent processors

10 agents setup		
Number of processors	Speedup	Efficiency
1	1.00000	1.00000
2	1.85166	0.92583
3	2.51535	0.83845
4	2.98958	0.74740
5	3.59168	0.71834
6	3.38780	0.56463
7	3.27178	0.46740
8	3.20883	0.40110
9	3.26418	0.36269
10	3.17654	0.31765

11 agents setup		
Number of processors	Speedup	Efficiency
1	1.00000	1.00000
2	1.71690	0.85845
3	2.46524	0.82175
4	3.05670	0.76417
5	3.29491	0.65898
6	3.29924	0.54987
7	3.29733	0.47105
8	3.15146	0.39393
9	3.10228	0.34470
10	3.08878	0.30888
11	3.10209	0.28201

12 agents setup		
Number of processors	Speedup	Efficiency
1	1.00000	1.00000
2	1.70365	0.85182
3	2.49313	0.83104
4	3.10946	0.77737
5	3.30414	0.66083
6	3.45204	0.57534
7	3.31583	0.47369
8	3.08166	0.38521
9	3.02821	0.33647
10	3.04780	0.30478
11	3.07456	0.27951
12	3.16562	0.26380

13 agents setup		
Number of processors	Speedup	Efficiency
1	1.00000	1.00000
2	1.68368	0.84184
3	2.31527	0.77176
4	3.01960	0.75490
5	3.36139	0.67228
6	3.34980	0.55830
7	3.22180	0.46026
8	3.14179	0.39272
9	2.88653	0.32073
10	3.05479	0.30548
11	2.97280	0.27025
12	2.98579	0.24882
13	3.16559	0.24351

14 agents setup		
Number of processors	Speedup	Efficiency
1	1.00000	1.00000
2	1.63738	0.81869
3	2.26342	0.75447
4	2.70845	0.67711
5	3.21424	0.64285
6	3.16180	0.52697
7	3.36398	0.48057
8	3.18328	0.39791
9	3.05676	0.33964
10	3.14229	0.31423
11	2.81919	0.25629
12	2.88520	0.24043
13	2.89885	0.22299
14	3.02550	0.21611

15 agents setup		
Number of processors	Speedup	Efficiency
1	1.00000	1.00000
2	1.57294	0.78647
3	2.25261	0.75087
4	2.79550	0.69888
5	3.31055	0.66211
6	3.32350	0.55392
7	3.25900	0.46557
8	3.26477	0.40810
9	2.96960	0.32996
10	2.89461	0.28946
11	2.90078	0.26371
12	2.84556	0.23713
13	2.87564	0.22120
14	2.90441	0.20746
15	2.96471	0.19765

16 agents setup		
Number of processors	Speedup	Efficiency
1	1.00000	1.00000
2	1.55449	0.77725
3	2.15604	0.71868
4	2.82989	0.70747
5	3.14860	0.62972
6	3.39482	0.56580
7	3.30918	0.47274
8	3.39829	0.42479
9	3.02453	0.33606
10	2.81450	0.28145
11	2.75354	0.25032
12	2.74618	0.22885
13	2.75238	0.21172
14	2.82771	0.20198
15	2.96907	0.19794
16	2.92251	0.18266

Appendix G : Table of results for speedup and efficiency of varying number of agents

1 Agent Processor				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 1 processor	Speedup for 1 processors = 1 processor time / 1 processor time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:27	1.00000	1.00000
2	00:00:56	00:00:56	1.00000	1.00000
3	00:01:24	00:01:24	1.00000	1.00000
4	00:01:55	00:01:55	1.00000	1.00000
5	00:02:11	00:02:11	1.00000	1.00000
6	00:02:40	00:02:40	1.00000	1.00000
7	00:03:00	00:03:00	1.00000	1.00000
8	00:03:23	00:03:23	1.00000	1.00000
9	00:03:45	00:03:45	1.00000	1.00000
10	00:04:13	00:04:13	1.00000	1.00000
11	00:04:30	00:04:30	1.00000	1.00000
12	00:04:49	00:04:49	1.00000	1.00000
13	00:05:05	00:05:05	1.00000	1.00000
14	00:05:19	00:05:19	1.00000	1.00000
15	00:05:32	00:05:32	1.00000	1.00000
16	00:05:53	00:05:53	1.00000	1.00000

2 Agent Processors				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 2 processors	Speedup for 2 processors = 1 processor time / 2 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:31	1.77756	0.88878
3	00:01:24	00:00:49	1.70778	0.85389
4	00:01:55	00:00:56	2.05155	1.02577
5	00:02:11	00:01:10	1.87230	0.93615
6	00:02:40	00:01:22	1.94533	0.97266
7	00:03:00	00:01:40	1.79799	0.89900
8	00:03:23	00:01:49	1.86662	0.93331
9	00:03:45	00:02:02	1.84008	0.92004
10	00:04:13	00:02:17	1.85166	0.92583
11	00:04:30	00:02:37	1.71690	0.85845
12	00:04:49	00:02:50	1.70365	0.85182
13	00:05:05	00:03:01	1.68368	0.84184
14	00:05:19	00:03:15	1.63738	0.81869
15	00:05:32	00:03:31	1.57294	0.78647
16	00:05:53	00:03:47	1.55449	0.77725

Appendix G : Table of results for speedup and efficiency of varying number of agents

3 Agent Processors				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 3 processors	Speedup for 3 processors = 1 processor time / 3 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:37	2.26541	0.75514
4	00:01:55	00:00:48	2.39743	0.79914
5	00:02:11	00:00:55	2.40135	0.80045
6	00:02:40	00:00:59	2.72326	0.90775
7	00:03:00	00:01:12	2.50582	0.83527
8	00:03:23	00:01:21	2.52070	0.84023
9	00:03:45	00:01:23	2.69372	0.89791
10	00:04:13	00:01:41	2.51535	0.83845
11	00:04:30	00:01:49	2.46524	0.82175
12	00:04:49	00:01:56	2.49313	0.83104
13	00:05:05	00:02:12	2.31527	0.77176
14	00:05:19	00:02:21	2.26342	0.75447
15	00:05:32	00:02:27	2.25261	0.75087
16	00:05:53	00:02:44	2.15604	0.71868

4 Agent Processors				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 4 processors	Speedup for 4 processors = 1 processor time / 4 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:00	#DIV/0!	#DIV/0!
4	00:01:55	00:00:42	2.70910	0.67727
5	00:02:11	00:00:49	2.65598	0.66400
6	00:02:40	00:00:58	2.75300	0.68825
7	00:03:00	00:01:02	2.90680	0.72670
8	00:03:23	00:01:03	3.24659	0.81165
9	00:03:45	00:01:13	3.05767	0.76442
10	00:04:13	00:01:25	2.98958	0.74740
11	00:04:30	00:01:28	3.05670	0.76417
12	00:04:49	00:01:33	3.10946	0.77737
13	00:05:05	00:01:41	3.01960	0.75490
14	00:05:19	00:01:58	2.70845	0.67711
15	00:05:32	00:01:59	2.79550	0.69888
16	00:05:53	00:02:05	2.82989	0.70747

5 Agent Processors				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 5 processors	Speedup for 5 processors = 1 processor time / 5 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:00	#DIV/0!	#DIV/0!
4	00:01:55	00:00:00	#DIV/0!	#DIV/0!
5	00:02:11	00:00:48	2.76265	0.55253
6	00:02:40	00:00:56	2.84389	0.56878
7	00:03:00	00:01:01	2.94106	0.58821
8	00:03:23	00:01:08	2.98226	0.59645
9	00:03:45	00:01:10	3.21845	0.64369
10	00:04:13	00:01:10	3.59168	0.71834
11	00:04:30	00:01:22	3.29491	0.65898
12	00:04:49	00:01:27	3.30414	0.66083
13	00:05:05	00:01:31	3.36139	0.67228
14	00:05:19	00:01:39	3.21424	0.64285
15	00:05:32	00:01:40	3.31055	0.66211
16	00:05:53	00:01:52	3.14860	0.62972

6 Agent Processors				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 6 processors	Speedup for 6 processors = 1 processor time / 6 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:00	#DIV/0!	#DIV/0!
4	00:01:55	00:00:00	#DIV/0!	#DIV/0!
5	00:02:11	00:00:00	#DIV/0!	#DIV/0!
6	00:02:40	00:00:51	3.10436	0.51739
7	00:03:00	00:01:00	2.99620	0.49937
8	00:03:23	00:01:03	3.23961	0.53993
9	00:03:45	00:01:09	3.27410	0.54568
10	00:04:13	00:01:15	3.38780	0.56463
11	00:04:30	00:01:22	3.29924	0.54987
12	00:04:49	00:01:24	3.45204	0.57534
13	00:05:05	00:01:31	3.34980	0.55830
14	00:05:19	00:01:41	3.16180	0.52697
15	00:05:32	00:01:40	3.32350	0.55392
16	00:05:53	00:01:44	3.39482	0.56580

Appendix G : Table of results for speedup and efficiency of varying number of agents

	7 Agent Processors			
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 7 processors	Speedup for 7 processors = 1 processor time / 7 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:00	#DIV/0!	#DIV/0!
4	00:01:55	00:00:00	#DIV/0!	#DIV/0!
5	00:02:11	00:00:00	#DIV/0!	#DIV/0!
6	00:02:40	00:00:00	#DIV/0!	#DIV/0!
7	00:03:00	00:00:58	3.12929	0.44704
8	00:03:23	00:01:03	3.21973	0.45996
9	00:03:45	00:01:12	3.13493	0.44785
10	00:04:13	00:01:17	3.27178	0.46740
11	00:04:30	00:01:22	3.29733	0.47105
12	00:04:49	00:01:27	3.31583	0.47369
13	00:05:05	00:01:35	3.22180	0.46026
14	00:05:19	00:01:35	3.36398	0.48057
15	00:05:32	00:01:42	3.25900	0.46557
16	00:05:53	00:01:47	3.30918	0.47274

	8 Agent Processors			
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 8 processors	Speedup for 8 processors = 1 processor time / 8 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:00	#DIV/0!	#DIV/0!
4	00:01:55	00:00:00	#DIV/0!	#DIV/0!
5	00:02:11	00:00:00	#DIV/0!	#DIV/0!
6	00:02:40	00:00:00	#DIV/0!	#DIV/0!
7	00:03:00	00:00:00	#DIV/0!	#DIV/0!
8	00:03:23	00:01:05	3.11208	0.38901
9	00:03:45	00:01:13	3.09520	0.38690
10	00:04:13	00:01:19	3.20883	0.40110
11	00:04:30	00:01:26	3.15146	0.39393
12	00:04:49	00:01:34	3.08166	0.38521
13	00:05:05	00:01:37	3.14179	0.39272
14	00:05:19	00:01:40	3.18328	0.39791
15	00:05:32	00:01:42	3.26477	0.40810
16	00:05:53	00:01:44	3.39829	0.42479

9 Agent Processors				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 9 processors	Speedup for 9 processors = 1 processor time / 9 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:00	#DIV/0!	#DIV/0!
4	00:01:55	00:00:00	#DIV/0!	#DIV/0!
5	00:02:11	00:00:00	#DIV/0!	#DIV/0!
6	00:02:40	00:00:00	#DIV/0!	#DIV/0!
7	00:03:00	00:00:00	#DIV/0!	#DIV/0!
8	00:03:23	00:00:00	#DIV/0!	#DIV/0!
9	00:03:45	00:01:08	3.32535	0.36948
10	00:04:13	00:01:18	3.26418	0.36269
11	00:04:30	00:01:27	3.10228	0.34470
12	00:04:49	00:01:35	3.02821	0.33647
13	00:05:05	00:01:46	2.88653	0.32073
14	00:05:19	00:01:44	3.05676	0.33964
15	00:05:32	00:01:52	2.96960	0.32996
16	00:05:53	00:01:57	3.02453	0.33606

10 Agent Processors				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 10 processors	Speedup for 10 processors = 1 processor time / 10 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:00	#DIV/0!	#DIV/0!
4	00:01:55	00:00:00	#DIV/0!	#DIV/0!
5	00:02:11	00:00:00	#DIV/0!	#DIV/0!
6	00:02:40	00:00:00	#DIV/0!	#DIV/0!
7	00:03:00	00:00:00	#DIV/0!	#DIV/0!
8	00:03:23	00:00:00	#DIV/0!	#DIV/0!
9	00:03:45	00:00:00	#DIV/0!	#DIV/0!
10	00:04:13	00:01:20	3.17654	0.31765
11	00:04:30	00:01:27	3.08878	0.30888
12	00:04:49	00:01:35	3.04780	0.30478
13	00:05:05	00:01:40	3.05479	0.30548
14	00:05:19	00:01:41	3.14229	0.31423
15	00:05:32	00:01:55	2.89461	0.28946
16	00:05:53	00:02:06	2.81450	0.28145

11 Agent Processors				
Number of agent KSs setup	Average time per move on 1 processor	Average time per move on 11 processors	Speedup for 11 processors = 1 processor time / 11 processors time	Efficiency = Speedup factor / number of processor used
1	00:00:27	00:00:00	#DIV/0!	#DIV/0!
2	00:00:56	00:00:00	#DIV/0!	#DIV/0!
3	00:01:24	00:00:00	#DIV/0!	#DIV/0!
4	00:01:55	00:00:00	#DIV/0!	#DIV/0!
5	00:02:11	00:00:00	#DIV/0!	#DIV/0!
6	00:02:40	00:00:00	#DIV/0!	#DIV/0!
7	00:03:00	00:00:00	#DIV/0!	#DIV/0!
8	00:03:23	00:00:00	#DIV/0!	#DIV/0!
9	00:03:45	00:00:00	#DIV/0!	#DIV/0!
10	00:04:13	00:00:00	#DIV/0!	#DIV/0!
11	00:04:30	00:01:27	3.10209	0.28201
12	00:04:49	00:01:34	3.07456	0.27951
13	00:05:05	00:01:43	2.97280	0.27025
14	00:05:19	00:01:53	2.81919	0.25629
15	00:05:32	00:01:54	2.90078	0.26371
16	00:05:53	00:02:08	2.75354	0.25032

Appendix H : emDARBS IPC implementation

In the following sections, the implementation of each of the classes in emDARBS IPC is explained along with its functions.

CkwSignalHandler Class

As explained earlier, SARNUX does not have communication interrupts in the form of signals as found in Linux and because of this, there is no need for a signal handler class. Therefore all the functions that are in the original LnSignalHandler class are now just empty functions. The reason for keeping this class is just for the sake of backward compatibility.

LnTcpClient Class (Wrapper Class)

This class is from original DARBS and is used by the KS clients to connect to the server and to transmit and receive messages from the server. The structure of the class (i.e. its member functions and member data) is kept the same just that there is a new implementation file for this class which is in CkwTcpClient.cpp file. The functions in this class and their description are explained as follows:

void LnTcpClient::LnTcpClient(void)
This is the constructor for the LnTcpClient class. The first thing that is done is to create an OSLinkToTcpClient class object and store the pointer to that object in m_arg member data. The last thing it does is to set the m_onSigIO and m_onSigPIPE member data to point to NULL.

void LnTcpClient::~LnTcpClient(void)
There is nothing done for the destructor of the LnTcpClient class as the SARNode is targeted to run in an embedded system and an embedded system usually runs indefinitely and should never stop. Therefore the destructor should never be called.

int LnTcpClient::connectTo(string address, int port)
This function is called by the KS client to make a connection to the BB server. The address and port number of the server to connect to is pass in as parameters to the function but this is not used. All that is done in this function is to pass control to OSLinkToTcpClient::StartOSLinkClient(). There was a change to the original DARBS source code that required this function to return the error value of the call to connect. So to match up with the new source code, this function now always returns the value 0 for a successful connection.

void LnTcpClient::closeConnection(void)
Like the destructor, this function is not intended to be called at all in emDARBS. Therefore this function does nothing.

void LnTcpClient::setCallback(int signo, void(*func)(void*,string,void*), void* arg)
This is actually an overloaded function as there are two function declarations for this function. In DARBS they are both used for setting the call back function to call when a signal (either SIGIO or SIGPIPE) has been triggered. But in SARNUX there is no signal

as such, so the SIGIO version of this function is used only (as this is the signal that would trigger when something is received from the TCP/IP socket). The other SIGPIPE signal is ignore and thus that function does nothing. In the SIGIO version of this function, the pointer to the call back function and the argument to be passed back to the call back function is pass down to OSLinkToTcpClient::setCallbackFunction() function.

void LnTcpClient::removeCallback(int signo)
Like the destructor, this function is not intended to be called at all in emDARBS. Therefore this function does nothing.

void LnTcpClient::sendData(string data)
This function is called to transmit a message/data to the server. The string of data to be transmitted is passed into this function as a parameter. This function would call the OSLinkToTcpClient::sendMessage() function passing in to it the pointer to the string of data. Only the pointer is passed in to sendMessage() function as this reduces the usage of memory and thus more efficient and faster.

void LnTcpClient::onSigio(int fd)
This is a virtual function from the derived LnSignalHandler class and is called from the LnSignalHandler class when a SIGIO signal has occurred. However as this does not apply in SARNUX case, this function is just a blank function. It is only here for the sake of backward compatibility.

void LnTcpClient::onSigpipe(int fd)
This function is the same as LnTcpClient::onSigio() function as is only here for backward compatibility. This is just a blank function.

OSLinkToTcpClient Class

This is a new class created as an interface between the LnTcpClient class and the OSLinkClientHandler class. As such there is an object of OSLinkClientHandler class created as a private member data of OSLinkToTcpClient class. Also created in this class is an object of CHAN (communication channel in SARNUX) which is used as an internal channel to transmit messages between the main KS client thread and the transmitting thread. Other static private member data of the OSLinkToTcpClient class are used to hold information on the status of the communication link. The OSLinkToTcpClient class contains similar functions like that of in LnTcpClient class except that they are implemented differently. The description of the functions in this class is as follows:

void OSLinkToTcpClient::OSLinkToTcpClient(void)
This constructor just call OSLinkToTcpClient::Initialise() which is a private function of this class.

void OSLinkToTcpClient::~OSLinkToTcpClient(void)
As this system is designed to be run on an embedded system, the destructor for this class should not be call. Therefore this is a blank function.

void OSLinkToTcpClient::Initialise(void)
This function is only called from the constructor of this class. This function just initialise the static private member data of this class to NULL and allocate a new string object and

storing the address of this object to the static private member, pMsgBuffer. The reason this initialisation is done in this function instead of in the constructor is because the constructor cannot initialise static private member data.

void OSLinkToTcpClient::StartOSLinkClient(void *LnTcpClientobj)
This function is called from LnTcpClient::connectTo(). The first thing that this function does is to store the pointer of the calling class (LnTcpClient) to a local private member data. Then it calls the OSLinkClientHandler::SetOnReceivedMsgFunction() to let the OSLinkClientHandler know which function to call when it receives a message from the BB. The parameter pass to the OSLinkClientHandler::SetOnReceivedMsgFunction() function is the address of OSLinkToTcpClient::onReceivedMsg() function. Therefore on receiving a message from the BB, the OSLinkToTcpClient::onReceivedMsg() function would be called. There are two reasons why this function is used instead of the function set in the setCallbackFunction(), one is because the required parameter passed to onReceivedMsg() function is different than the function set in setCallbackFunction(). The second reason is that there is no guarantee that the main KS client program would call the LnTcpClient::setCallback() function before calling the connectTo() function. This is to cover the small time frame of starting the receiver without the receiving function being set. This is thought to be important in an embedded system. Finally this OSLinkToTcpClient::StartOSLinkClient() function tries to start the transmitter and receiver handler in the OSLinkClientHandler class. This is done by calling the OSLinkClientHandler::StartRxTxHandler() function with the pointer to the CHAN object. The CHAN object would be used to communicate with the TxHandler thread.

void OSLinkToTcpClient::sendMessage(string *pMessage)
This function is called from LnTcpClient::sendData() and it is use to send a message to the TxHandler thread to transmit the message to the BB server. So this function just passes the message to be transmitted to the BB server to the TxHandler thread via the KS_To_TxHandler CHAN object. This is done in two steps, the first is to transmit the length of the message to the TxHandler thread, then only transmit the actual message to the TxHandler thread.

void OSLinkToTcpClient::setCallbackFunction(void(*pFunc)(void*, string,void*), void *passBackArg)
This function is called from the LnTcpClient::setCallback() and all this function does is to store the pointer to the call back function and the argument to be pass back to the call back function in the local static private member data.

string OSLinkToTcpClient::lastMessage(void)
This is more of a debug function and it is used to return the message that was last received to the calling function. At the moment this function is not use.

void OSLinkToTcpClient::onReceivedMsg(word *pRxMsg)
This function is called when the RxHandler thread receives a message from the BB server. The first thing that this function does is to convert the received message into a string object and the store it in the static private member data, pMsgBuffer. The next thing it does is to check if the call back function has been set-up or not. If it has been set-up, then it calls the call back function passing to it the pointer to the LnTcpClient object, the string of the message received and the user argument that was set-up earlier in the

setCallbackFunction(). If there has been no call back function set-up then an error would be generated.

OSLinkClientHandler Class

This is the actual class created to handle the communications of SARNode. This class contains the low level functions to do with communications and the starting up of the TxHandler and RxHandler threads. It also contains the implementation of both the TxHandler and RxHandler threads. The description of the functions in this class is as follows:

void OSLinkClientHandler::OSLinkClientHandler(void)
This constructor just call OSLinkClientHandler::Initialise() function which is a private function of this class.

void OSLinkClientHandler::~OSLinkClientHandler(void)
As this system is designed to be run on an embedded system, the destructor for this class should not be call. Therefore this is a blank function.

void OSLinkClientHandler::Initialise(void)
This function is only called from the constructor of this class. The only thing that this function does is to set the static private member data, onReceivedMsg to NULL. The onReceivedMsg member data is used to hold the address of the function to call when a message is receive from the BB.

bool OSLinkClientHandler::StartRxTxHandler(CHAN *KS_To_Tx)
This function is called from OSLinkToTcpClient::StartOSLinkClient(). This function will start the RxHandler thread and the TxHandler thread. The first thread it creates is the TxHandler thread passing to it the pointer to the internal channel to use to communicate with the main KS client thread. The next thread it creates is the RxHandler thread. Failure to create either one of these threads would cause this function to return a false; otherwise a true is return on the success of creating both this thread.

void OSLinkClientHandler::SetOnReceivedMsgFunction(void(*pFunction)(word *pRxMsg))
This function is used to set the pointer to the function to call when a message is receive from the BB. This pointer is stored in the static private member data, onReceivedMsg.

void OSLinkClientHandler::RxHandler(void)
This function is the start of the RxHandler thread. The purpose of this thread is to monitor the OS-Link receiver indefinitely for a message from the BB and when a message is received, it calls the onReceivedMsg function (if it is set-up). The first thing that this function does is to allocate a message buffer area of MAX_RX_SIZE. Then it initialises the OS-Link receiving channel to the message ID (MID) set in SARNet map table, SARNet_Map_KS. Finally it goes into a forever loop, waiting for a message from the BB and when a message is received from the BB, it calls the onReceivedMsg function and then it goes back to wait for the next message from the BB.

void OSLinkClientHandler::TxHandler(CHAN *internal_KS_TO_TX)
This function is the start of the TxHandler thread. The main purpose of this thread is to sit forever waiting for a message to be transmitted to the BB from the main KS client

thread. The main KS client thread will pass the message to be transmitted to the BB to this TxHandler thread and it will then transmit this message to the BB via the OS-Link transmitter channel. The first thing that this function does is to allocate a message buffer of MAX_TX_SIZE. Then it initialises the OS-Link transmitter channel with the routing header set in the SARNet map table, SARNet_Map_KS. It then goes into a forever loop. In this loop, the first thing it does is to wait for a message length to be transmitted to it from the main KS client thread. The length of a message is set to be four bytes long, i.e. from 1 to 232 long (zero is not a valid length to be transmitted). With the length known, the TxHandler would check to make sure that the length received is not zero and less than MAX_TX_SIZE, otherwise an error would be thrown. With the length known to be within the acceptable limits, the next thing that the TxHandler does is to receive the actual message from the main KS client thread and store it in the message buffer allocated earlier. Finally it transmits this message to the BB and goes back to listening for a message length from the main KS client thread.

LnTcpServer Class (Wrapper Class)

This class is from original DARBS and is used by the BB server to start up the server to listen and wait for a transmission from the KS clients. The structure of the class (i.e. its member functions and member data) is kept the same just that there is a new implementation file for this class which is in CkwTcpServer.cpp file. The functions in this class and their description are explained as follows:

void LnTcpServer::LnTcpServer(void)
This is the constructor for the LnTcpServer class. The first thing that is done is to create an OSLinkToTcpServer class object and store the pointer to that object in m_arg member data. The last thing it does is to set the m_onSigIO and m_onSigPIPE member data to point to NULL.

void LnTcpServer::~LnTcpServer(void)
This destructor is an empty function as in an embedded system this function should never be call.

void LnTcpServer::openPort(int port)
This function is called to start the server up and make it ready to receive transmission from the clients. As this function was initially designed to work on TCP/IP protocol, the control is now pass to OSLinkToTcpServer::StartOSLinkServer() function instead.

void LnTcpServer::closePort(void)
This function was initially used to close the BB server down but in an embedded system this would not happen and therefore this function is just a blank function.

string LnTcpServer::getBuffer(void)
This is a debugging function used to return the last message received by the server. In this function, the control is pass to OSLinkToTcpServer::lastMessage() function.

void LnTcpServer::setCallback(int signo, void(*func)(void*,string,void*), void* arg)
This is actually an overloaded function as there are two function declarations for this function. In the original DARBS software they are both used for setting the call back function to call when a signal (either SIGIO or SIGPIPE) has been triggered. But in SARNUX there is no signal as such, so the SIGIO version of this function is used only

(as this is the signal that would trigger when something is received from the TCP/IP socket). The other SIGPIPE signal is ignore and thus that function does nothing. In the SIGIO version of this function, the pointer to the call back function and the argument to be passed back to the call back function is pass down to OSLinkToTcpServer::setCallbackFunction() function.

void LnTcpServer::removeCallback(int signo)
As in an embedded system it is assumed that the call back function once registered would never be removed and therefore this function is a blank function.

void LnTcpServer::onSigio(int fd)
The description is the same as LnTcpClient::onSigio().

void LnTcpServer::sendData(string message)
This function is called by the BB to transmit a message back to the current calling client (i.e. the sender of the last received message). To do this the OSLinkToTcpServer::sendMessage() function is called with the pointer to the message to be transmitted.

void LnTcpServer::broadcastMessage(string message)
This function is called by the BB to broadcast a message to all the KS clients except the current calling client (i.e. the sender of the last received message). This is done by calling the OSLinkToTcpServer::broadcastMessage() function.

void LnTcpServer::onSigpipe(int fd)
The description is the same as LnTcpClient::onSigpipe().

OSLinkToTcpServer Class

This class acts as an interface between the LnTcpServer class and the OSLinkServerHandler class. As such there is an object of OSLinkServerHandler class in this interface class as was shown in Figure 51. Similar to the OSLinkToTcpClient class, this class uses static private member data to hold the status of the current communication link. However unlike the OSLinkToTcpClient, this class has an array of internal communication channels for communicating with the TxHandler threads. It also has a SEMA pointer object (a semaphore provided by SARNUX) which is used to make sure that the access to the BB is done with mutual exclusion, i.e. only one KS client can access the BB at a time and there rest is queued on a first in first out (FIFO) basis. The description of the functions in this class is as follows:

void OSLinkToTcpServer::OSLinkToTcpServer(void)
In this constructor, the first thing that is done is to set the SEMA pointer object to NULL and then to call the OSLinkToTcpServer::Initialise() function.

void OSLinkToTcpServer::~OSLinkToTcpServer(void)
As the destructor is not meant to be used in an embedded system, therefore this is a blank function.

void OSLinkToTcpServer::Initialise(void)

This function is only called from the constructor and it is used to set all the static private member data to NULL and to allocate a string class object for use later. Static private member data, pMsgBuffer is used to store the pointer to the string object.

void OSLinkToTcpServer::StartOSLinkServer(void *LnTcpServerobj)
This function is called from LnTcpServer::openPort() function and it is used to start all RxHandler threads and TxHandler threads. The first thing that this function does is to store the pointer to the calling LnTcpServer class object in a private member data. Then it allocates a semaphore (SEMA object) and tries to initialise it. This semaphore is used to provide mutual exclusion access to the BB from the RxHandler threads. This means that only one RxHandler thread can call the BB server services at any one time. Then it sets up the call back function to call when any one of the RxHandler thread receives a message. Finally, it calls the OSLinkServerHandler::StartRxTxHandler() function *n* times, where *n* is the number of KS client in the system. This means that there would be a pair of RxHandler and TxHandler thread for each of the KS client in the system.

void OSLinkToTcpServer::sendMessage(string *pMessage)
This function is call from LnTcpServer::sendData() and it is used by the BB to send a message back to the current calling client. This function would transmit to the TxHandler thread of the current calling client the length of the message and then the actual message via the internal communication channel.

void OSLinkToTcpServer::broadcastMessage(string *pMessage)
This function is call from LnTcpServer::broadcastMessage() and it is used by the BB to send a message to all the KS clients in the system except the current calling client. The workings of this function is similar to sendMessage() except that this is done in a loop of *n* times, where *n* is the number of KS in the system and with the exception of not transmitting to the current calling client.

void OSLinkToTcpServer::setCallbackFunction(void(*pFunc)(void*, string,void*), void *passBackArg)
This function is used to set the pointer to the call back function to call when a message is received from any one of the KS clients.

string OSLinkToTcpServer::lastMessage(void)
This is a debug function and it is used to return the last received message which is from the current calling client.

void OSLinkToTcpServer::onReceivedMsg(word *pRxMsg, int KS)
This is the call back function that would be call when a messaged is received from a KS. This function is called with the assumption that the semaphore (SEMA object) is used to guarantee mutual exclusion access of this function. The first thing that this function does is to store a copy of the message received to the private member data so that the lastMessage() function can use it when called. The next thing to do is to store the KS client number of the sender of the message. This is the current calling client number. Then it checks to see if the call back function has been set or not, if set it will call the call back function otherwise it will throw up an error.

OSLinkServerHandler Class

This is the actual class that handles the OS-Link communications for the BB server. The overall function of this class is similar to OSLinkClientHandler class except that this class expects the creation of a set of RxHandler and TxHandler threads for each KS client in the system. The description of the functions in this class is as follows:

void OSLinkServerHandler::OSLinkServerHandler(void)
This is the constructor of the OSLinkServerHandler class and all it does is to call the Initialise() function to initialise the private member data.

void OSLinkServerHandler::~OSLinkServerHandler(void)
As this system is targeted for the embedded system, there should not be a call for the destructor of this class. As such, this function is a blank function.

void OSLinkServerHandler::Initialise(void)
This function is only called from the constructor and its function is to initialise the call back function pointer to NULL.

bool OSLinkServerHandler::StartRxTxHandler(SEMA *pRxSemaphore, CHAN *BB_To_Tx, int KS)
This function is called from OSLinkToTcpServer::StartOSLinkServer() and is used to start a pair of RxHandler and TxHandler threads. It will try to start TxHandler thread and then RxHandler thread, if either one of these threads fails to start, then a FALSE is returned from this function, otherwise a TRUE is returned. For the TxHandler thread, two parameters are passed to it on start up, which are the KS number and the pointer to the internal channel to use for receiving messages from the BB. For the RxHandler thread, another two parameters are passed to it on start up, which are the KS number and the pointer to the semaphore to use before calling the call back function.

void OSLinkServerHandler::SetOnReceivedMsgFunction(void(*pFunction)(word *pRxMsg, int KS))
This function is used to set the pointer to the call back function to use when a message is received from the KS client.

void OSLinkServerHandler::RxHandler(int KS, SEMA *pSemaphore)
This function is the start of the RxHandler thread. The purpose of this thread is to monitor the OS-Link receiver indefinitely for a message from the KS client that this particular RxHandler thread is suppose to handle. The KS client that a particular RxHandler thread is supposed to handle is passed in to this thread as a parameter called KS. When a message is received from the KS client that the RxHandler is monitoring, it calls and waits on the semaphore. On returning from the semaphore (this means that the critical resource is free), the onReceivedMsg function (if it is set-up) is called. On the start of this thread, the first thing that it does is to allocate a message buffer area of MAX_RX_SIZE. Then it initialises the OS-Link receiving channel to the message ID (MID) of this particular KS number set in SARNet map table, SARNet_Map_BB. Finally it goes into a forever loop, waiting for a message from the KS client and when a message is received from the KS client, it waits for the semaphore and then calls the onReceivedMsg function. Coming back from the onReceivedMsg function, the RxHandler goes back to wait for the next message from the KS client.

void OSLinkServerHandler::TxHandler(int KS, CHAN *internal_BB_TO_TX)
This function is the start of the TxHandler thread. The main purpose of this thread is to sit forever waiting for a message to be transmitted to the particular KS client (KS number is passed in as a parameter on the start of the TxHandler thread) that it is handling from the BB. The main BB thread will pass the message to be transmitted to the KS client to this TxHandler thread. The TxHandler thread will then transmit this message to the KS client via the OS-Link transmitter channel. The first thing that this function does is to allocate a message buffer of MAX_TX_SIZE. Then it initialises the OS-Link transmitter channel with the routing header set in the SARNet map table, SARNet_Map_BB. It then goes into a forever loop. In this loop, the first thing it does is to wait for a message length to be transmitted to it from the BB thread. The length of a message is set to be four bytes long, i.e. from 1 to 232 long (zero is not a valid length to be transmitted). With the length known, the TxHandler would check to make sure that the length received is not zero and less than MAX_TX_SIZE, otherwise an error would be thrown. With the length known to be within the acceptable limits, the next thing that the TxHandler does is to receive the actual message from the BB thread and store it in the message buffer allocated earlier. Finally it transmits this message to the KS and goes back to listening for a message length from the BB thread.

Appendix I : emDARBS TestCompare KS and output listing

TestCompare.dkf

```
/* max file size is 64K */
KS KS_Test_Compare

      KS_TYPE rule_based_KS

      INFERENCE_MODE MI_Forwardchain

      RULES
      [
            SetDataCompare
            AND
            TestCompare
      ]

      FIRABILITY_FLAG true

      IF
      [
            [not_on_partition [KS_Test_Compare is fired] ControlChars]
      ]

      THEN
      [
            [add [KS_Test_Compare is fired] ControlChars]
      ]

END
```

SetDataCompare.drf

```
RULE SetDataCompare

      IF
      [
            [not_on_partition [Set Data Compare Rule is fired]
ControlChars]
      ]

      THEN
      [
            [add [Test is 1] par1]
            AND
            [add [Test is 3] par1]
            AND
            [add [Test is 5] par1]
            AND
            [add [Test is 10] par1]
            AND
            [add [Test is 15] par1]
            AND
            [add [Set Data Compare Rule is fired] ControlChars]
      ]
```

```
      BECAUSE
            [Set Data Compare Rule is not fired]
END
```

TestCompare.drf

```
RULE TestCompare

      IF
      [
            [not_on_partition [TestCompare Rule is fired] ControlChars]
            AND
            [
            [on_partition [Test is ?num] par1]
            AND
            [compare [~num LessThan 10] ]
            ]
      ]

      THEN
      [
            [report [~num LessThan 10]]
            AND
            [add [TestCompare Rule is fired] ControlChars]
      ]

      BECAUSE
            [TestCompare Rule is not fired]
END
```

Output from blackboard server

```
DARBS Blackboard Server version 2.70
Compiled on Mar 23 2004 at 11:55:13
IP: 127.0.0.1 at Port: 9734
==================================

server received      : not_on_partition [KS_Test_Compare is fired]
ControlChars
answer from board : true

server received    : not_on_partition [Set Data Compare Rule is fired]
ControlChars
answer from board : true

server received   : add [Test is 1] par1
answer from board : partition par1 changed!

server received   : add [Test is 3] par1
answer from board : partition par1 changed!

server received   : add [Test is 5] par1
answer from board : partition par1 changed!

server received   : add [Test is 10] par1
answer from board : partition par1 changed!
```

```
server received   : add [Test is 15] par1
answer from board : partition par1 changed!

server received   : add [Set Data Compare Rule is fired] ControlChars
answer from board : partition ControlChars changed!

server received     : not_on_partition [TestCompare Rule is fired]
ControlChars
answer from board : true

server received   : ret_all [Test is ?num] par1
answer from board : true [ [ 1 ] [ 3 ] [ 5 ] [ 10 ] [ 15 ] ]

server received   : add [TestCompare Rule is fired] ControlChars
answer from board : partition ControlChars changed!

server received   : add [TestCompare Rule is fired] ControlChars
answer from board : partition ControlChars changed!

server received   : add [TestCompare Rule is fired] ControlChars
answer from board : partition ControlChars changed!

server received   : add [KS_Test_Compare is fired] ControlChars
answer from board : partition ControlChars changed!
```

Output from KS client

```
[1 LessThan 10]

[3 LessThan 10]

[5 LessThan 10]
```

Appendix J : Source code and publications in CD-ROM