Int. Conf. on Software Engineering Research and Practice (SERP'03), Las Vegas, June 2003, pp239-245

Design and Implementation of an Inter-Process Communication Model for an Embedded Distributed Processing Network

K.W. Choy, A.A. Hopgood, L. Nolle, B.C. O'Neill
The Nottingham Trent University
School of Computing and Mathematics,
Burton Street,
Nottingham NG1 4BU
United Kingdom
{kw.choy, adrian.hopgood, lars.nolle, brian.oneill}@ntu.ac.uk

Abstract

A network of embedded processors has been used as the platform for a distributed blackboard system, which is an artificial intelligence (AI) technique. A distributed blackboard system called DARBS had been created for a PC-based Linux operating system using the TCP/IP communication protocol. The Nottingham Trent University had developed an embedded distributed processing network called SARNets that can be used to run an embedded version of DARBS (emDARBS). SARNet uses the SARNUX operating system that uses the communicating sequential process (CSP) communication model. Because this communication model is different from that of the Linux operating system, a new Inter-Process Communication model was required that would emulate Linux's communication model on SARNUX. This paper focuses on the development of an Inter-Process Communication model designed to allow DARBS to run on the SARNet with minimum changes to the original DARBS source code.

Keywords: Embedded Systems, Inter-process Communication Model, Distributed Processing Network, Blackboard Systems, SARNet

1. Introduction

The emDARBS (embedded DARBS) comprises DARBS (Distributed Algorithmic & Rule-based Blackboard System) software running on the SARNet hardware. The aim of this work was to port DARBS from a PC-based Linux operating system to the SARNet environment using the SARNUX operating system [10]. This would enable the use of DARBS in an embedded distributed processing system. Since Linux's communication model and SARNUX's communication model are completely different, an Inter-Process Communication model was required.

1.1. DARBS

DARBS is a distributed blackboard system developed by the Open University and the Nottingham Trent University [1]. The original non-distributed blackboard system was called ARBS and was run on a single processor machine. A blackboard system is an artificial intelligence (AI) technique that is analogous to a team of experts who communicate their ideas by writing them on a blackboard [2]. The experts are represented by sets of rules, conventional procedures, neural networks, or other program modules. These modules are termed knowledge sources (KS). The blackboard (BB) is an area of global memory containing evolving information. The system's current state of understanding of a problem is stored here as it develops from a set of data towards a conclusion.

DARBS was selected for this work as it is a research-based distributed blackboard system that does not have a dedicated control module. Commercially available distributed blackboard systems were not considered owing to their cost and the limitations on access to their source code.

In DARBS, the blackboard system is modelled on the client/server model where the BB is the server and the KSs are the clients. Therefore, DARBS can be seen as a distributed blackboard system with the BB server running on one PC and other KS clients running on different PCs. This allows different KSs to run in parallel and thus to be truly opportunistic [2]. For this reason, DARBS may also be considered as a multi-agent collaborative system [3]. Different KS clients on different PCs can work independently and different solutions to a problem can be developed at the same time. It was therefore envisaged that the porting of DARBS to the embedded distributed processing network would enable the development of an embedded distributed intelligent system.

1.2. SARNet

The Parallel Processing Research Group at the Nottingham Trent University had developed embedded processor nodes (SARNodes) using StrongARM RISC processors for creating a network of distributed parallel processing units [4]. The SARNode was specially designed with a parallel embedded system in mind [5] and therefore has a low power consumption for a processor of its class [6]. Each SARNode consists of a StrongARM processor, SDRAM, and an OS-Link (over sampling link) [7] communication module. The SARNet is a network of SARNodes connected together via an ICR C416 router [5] which uses the OS-Link communication protocol. This OS-Link uses the wormhole routing strategy, which has a low overhead [5][8]. The SARNet can have many different network configurations, one of which is as shown in Figure 1. 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 [9] as shown in Figure 1. The operating system for the SARNode is SARNUX, which is designed for use in an embedded system [10].

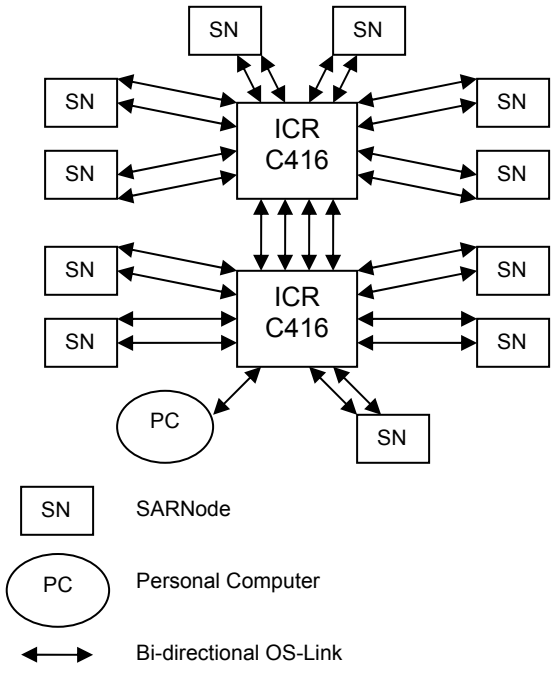

Figure 1. Possible ICR C416 network layout

1.3. Porting difficulties

The main difficulty of porting DARBS to the SARNet environment lies in the communication model. The DARBS communication module uses the Linux operating system's inter-process communication (IPC) model that, in turn, uses the TCP/IP communication protocol. 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 [11]. For example, the signal that is generated when a message is received on Linux's IPC model is SIGIO [11]. So by registering a function that is triggered by a message from the communication channel to the SIGIO signal, this function would 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. 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 the call would automatically pass 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 communication classes make use of these functions provided by Linux to implement the client/server model. The server's communication 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).

In SARNUX, however, the communication model is based on the communicating sequential process (CSP) model [12]. 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 unidirectional 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 [13]. If a process that 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.

This can easily lead to a deadlock situation [14]. To prevent this from happening, receiving buffers are used. However if the receiving buffers were full then the link would be blocked. It is up to the application designer to make sure that this does not happen.

Because of the two different communication models used by SARNUX and Linux, a new emDARBS Inter-Process Communication (IPC) model had to be designed that would minimise the changes required to the core DARBS source code.

2. Design and implementation of emDARBS IPC model

As the communications of DARBS are mainly between separate processors, the focus of the Inter-Process Communication (IPC) model is on external communications. To communicate between two threads running on different SARNodes, a routing header and message ID are required. The routing header is defined by the location on the network of the particular SARNode. The message ID (MID) is a unique number that identifies a particular message for a particular SARNode.

In the original DARBS, the BB server and the KS clients were designed to run as processes of their own. There are three classes in DARBS to handle the communication between processes: LnSignalHandler, LnTcpClient, and LnTcpServer. LnSignalHandler is the base class that handles the communication signals (interrupts) that comes from the Linux operating system. LnTcpClient is a derived class from the LnSignalHandler class and it is used to handle the client part of the TCP/IP communications. The LnTcpServer class (also derived from the LnSignalHandler class) handles the server side of the TCP/IP communications and is used by the server part of DARBS (i.e. the BB server).

2.1. Network layout and routing

In order for communication to take place in the SARNet network, 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. SARNet was designed to run as an embedded system and therefore all the network layout and settings are static and known at compile time. The layout of the SARNet network for emDARBS is defined in Figure 2. Only four SARNodes are shown in Figure 2, but in fact many more can be connected to the SARNet using the ICR C416 router [15]. For clarity, the router is not shown. The port number associated with each SARNode is as shown in Figure 2 along with the message ID used for each communication channel. The port number of the sender of a message is not important and therefore every message could originate from either one of the two ports available on the SARNode. For simplicity, Figure 2 only shows one of the ports that could originate a message. For example, the BB server can transmit a message to KS1 via its transmitting channel using MID 0xA (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.

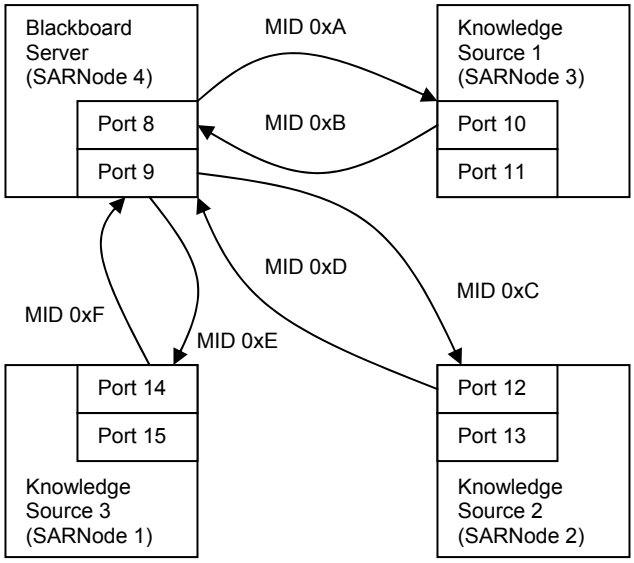

Figure 2. SARNet network layout for emDARBS

2.2. IPC class structure

The original DARBS had three classes to handle the TCP/IP communications: LnSignalHandler, LnTcpClient, and LnTcpServer. For compatibility, the original DARBS communication classes were kept and they now act as wrapper classes. To emulate Linux's communication model on the SARNUX, four new classes were created: two for the client side and two for the server side. An overall picture of the emDARBS IPC class structure using the UML standard is as shown in Figure 3. This figure shows that there are two sides of the communication model: the client side (LnTcpClient, OSLinkToTcpClient, and OSLinkClientHandler classes) and the server side (LnTcpServer, OSLinkToTcpServer, and OSLinkServerHandler classes). As the original DARBS communication classes are now 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 SARNUX's communication model. This would be handled by OSLinkToTcpClient and OSLinkToTcpServer classes

respectively. These two classes would 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. The design and implementation of both these sides are explained in the following sections.

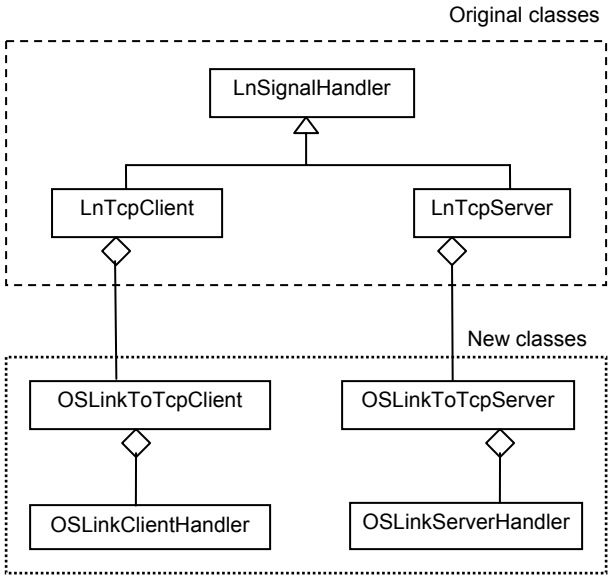

Figure 3. emDARBS IPC class structure

2.3. Client side IPC model

From Figure 3, it can be seen that the client side of the emDARBS IPC model consists of a new LnTcpClient (which now only acts as a wrapper class to hide the new classes from the application), OSLinkToTcpClient, and OSLinkClientHandler classes. The main functions required from the emDARBS IPC model for the client side are transmit, receive, connect to server, and set up the call-back function. Due to the block send and receive nature of the CSP model used in SARNUX, the only way for the main KS client program to continue running and still have the communication running in the background is to have separate threads running concurrently for receiving and transmitting. With multiple threads running, some form of synchronization and communication is required between the threads. The KS client side of emDARBS IPC solves this problem by the design shown in Figure 4.

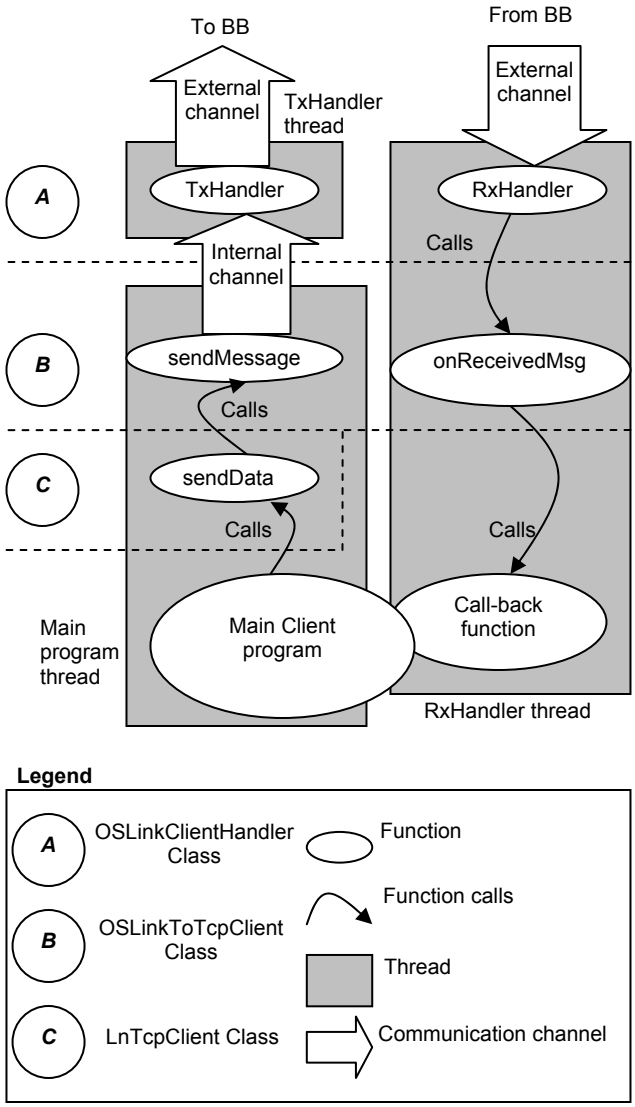

Figure 4. emDARBS IPC model of the client side

As shown in Figure 4, there are three threads running concurrently: TxHandler, RxHandler, and the main client program. Communication between threads is accomplished via an internal channel between the sendMessage function of the main client program thread and the TxHandler thread. As there are three threads running concurrently in the SARNode, it is possible for deadlock to occur [16]. Therefore access to common shared resources between the three threads is protected by masking the interrupts. This means that when a thread is using a common shared resource, the thread is run at a higher priority and no other threads can interrupt it while it is still using the common shared resource. When the thread has finished using the common shared resource,

the interrupts are unmasked and the priority of the thread is returned to its original level.

The purpose of the RxHandler thread is to wait for a message from the BB and then pass that message on to the onReceivedMsg function and then to the actual call-back function of the main client program. The main client program has to provide a function that is to be called when a message is received and the pointer to that function is passed on to the OSLinkToTcpClient class. This set-up must be done prior to the receipt of a message from the BB. It is up to the main client program 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 for the main client program to monitor the shared variables for changes made by the call-back function. The classes to which each of the function belongs can also be seen in Figure 4 (represented by the *A*, *B*, and *C* regions) and this gives an overview of the layout of the classes in the client side of the emDARBS IPC model.

2.4. Server side IPC model

For the server side, the IPC model is 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 main functions that the emDARBS IPC model for the server side is required to provide are open listening port, receive from multiple sources, transmit to single source, broadcast to multiple source, and set up the call-back function. Figure 5 shows an overview of the function of the transmitting side of the BB server.

As can be seen in Figure 5, there is a TxHandler thread dedicated for each KS client in the system. There would be *n* TxHandler threads where *n* is the number of KS clients in the system. It is implemented this way so that KSs on the top cannot block or slow down the KSs on the bottom from receiving a broadcast message. This is because the communication in the SARNUX operating system is implemented as a block receive and block transmit type. 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 5 are the classes to which each of the functions belongs. These are represented by the *A*, *B*, and *C* regions in Figure 5.

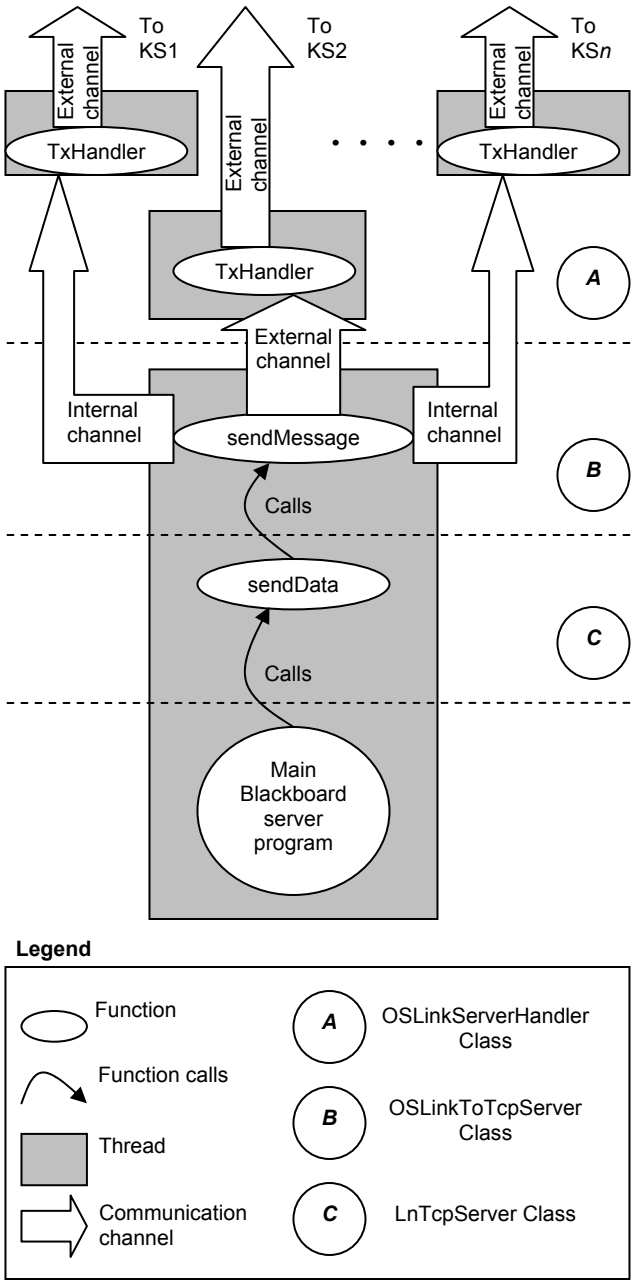

Figure 5. emDARBS IPC model for the transmitting side of the server

For the receiver side, the server needs to make sure that the messages coming from each of the KS clients are not missed but at the same time it must be ensured that the integrity of the data on the blackboard server is not compromised. To achieve this, access to the blackboard server has to be mutually exclusive [17], meaning that only one of the KS clients can access the BB server at any one time. Figure 6 shows how this is accomplished in the emDARBS IPC model.

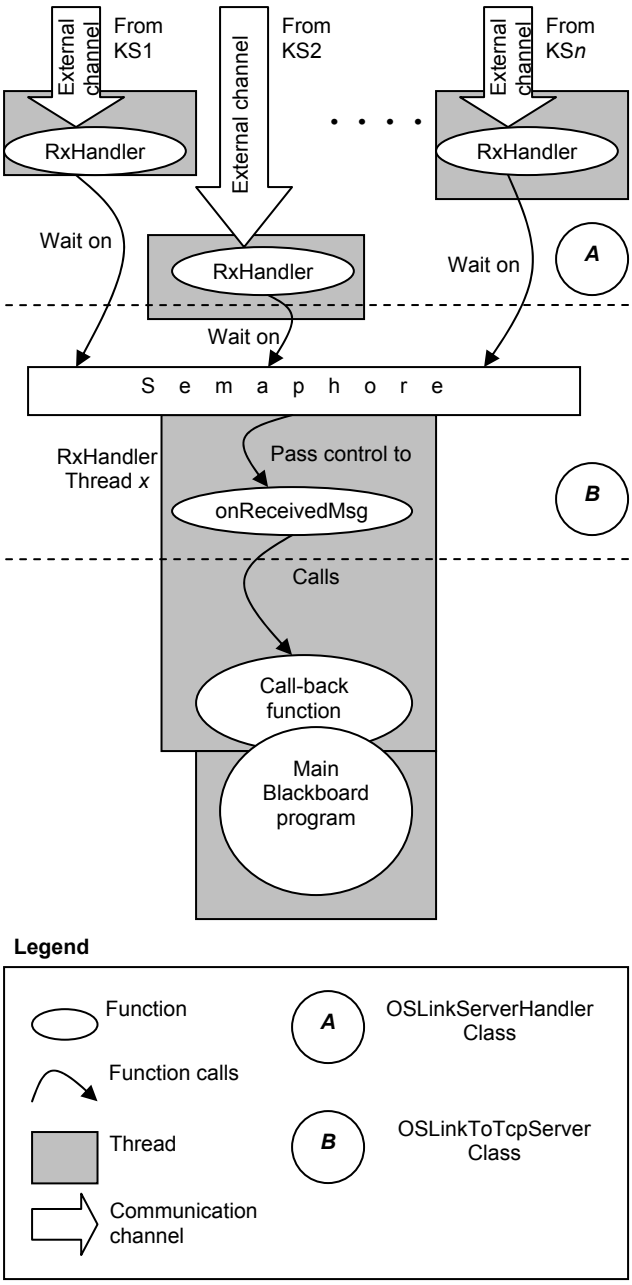

Figure 6. emDARBS IPC model for the receiving side of the server

As can be seen in Figure 6, there is an RxHandler thread allocated for each KS client in the system. There would be *n* RxHandler threads for *n* KS clients 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 for the semaphore [17]. If there is another thread already accessing the shared resource (in this case onReceivedMsg function) then the RxHandler would have to wait until the thread using the shared resource 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 queuing method implemented on the semaphore is of the First In First Out (FIFO) method.

The rest of the function from onReceivedMsg function onwards is similar to the client's receiver side. The thread on which the onReceivedMsg function and the call-back function would be running is determined by the RxHandler thread that has the semaphore. For example, suppose the RxHandler thread for KS1 has received a message from KS1 and then managed to obtain the semaphore first, then the thread that is running the onReceivedMsg function is the RxHandler thread for KS1. However, if the RxHandler thread for KS3 was slightly earlier in requesting the semaphore, then the RxHandler thread for KS1 would be blocked and the RxHandler thread for KS3 would go on and execute the onReceivedMsg function. By using the semaphore in the receiver side, mutual exclusion access to the blackboard is guaranteed and at the same time the multiple RxHandler threads that are executing concurrently guarantees that the BB server does not miss any messages coming from the KS clients.

3. Testing and validation

After implementing the emDARBS IPC model, the original DARBS terminal program was used to test the new emDARBS IPC model. The clients in the DARBS terminal program are terminal clients which take in commands from the user and send them to the BB server. The BB server was the full running BB server used in DARBS. The test for the new emDARBS IPC model was conducted in three parts, explained in the following paragraphs.

In the first part of the test, one client and one blackboard server were run on two SARNodes respectively (SARNode 4 and SARNode 3 of Figure 2). The result from this test was that a partition [1] could be created and data could be added to the partition on the blackboard via the terminal client. Also the blackboard did transmit back to the terminal client the contents of the partition. This proved that the emDARBS IPC can transmit and receive messages.

In the second part of the test, all four SARNodes were used, i.e. one blackboard server, and three terminal clients (SARNode 1, SARNode 2, SARNode 3, and SARNode 4 of Figure 2). The new IPC model functioned as it was design to during this test. All the clients received the broadcast message from the BB server correctly and could issue commands to the BB server correctly. This proved

that emDARBS IPC can handle multiple clients transmitting and receiving messages.

In the third part of the test, each of the three clients was programmed to automatically send requests to create a partition and to add 50 different data items to its partition. The clients were started simultaneously and the BB server started to handle the multiple requests from the clients. The result of this test was that the BB server handled all the requests properly according to the specifications. This proved that the semaphore approach worked and that the data integrity of the BB server was not compromised.

As a whole, all three tests proved that the emDARBS IPC model worked. emDARBS IPC can handle multiple clients transmitting and receiving messages simultaneously. Only minimum changes were required to the original DARBS code to make it work in the SARNet environment.

4. Conclusion and future work

The emDARBS IPC model worked and achieved its design aims, i.e. to enable the original DARBS to work in the new SARNet environment with minimum changes to the original DARBS source code. The emDARBS IPC model enabled DARBS to behave as though it was still using the TCP/IP communication protocol, but in fact it now uses the OS-Link communication protocol.

This work has demonstrated the use of a network of embedded processors (SARNet) in an artificial intelligence application. The architecture of the SARNet is aimed at users who require dedicated processors with good performance and low power for specific tasks. Each processor can be readily customised to suit the particular task. In this case, the SARNodes have been customised to run the KS clients and the BB server.

The design of the emDARBS IPC has enabled future work on fully parallelizing the BB server whereby the semaphore could be used to protect individual partitions instead of the whole BB database. This means that KS clients that are only accessing one partition of the BB would allow other KS clients to access other partitions of the BB concurrently so long as they do not access the same partition. The full DARBS software running in the embedded distributed processing network will represent a new kind of intelligent embedded system.

References

[1] 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, pp 161–170, 2001.

[2] Engelmore, R., Morgan, T., ed. "*Blackboard Systems*". Great Britain: Addison Wesley, 1988. pp. 2-15.

[3] Jennings, N. R., Wooldridge, M. J., ed. "*Agent Technology: Foundations, Applications, and Markets"*. Germany: Springer, 1998. pp. 32-34.

[4] 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.

[5] 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.

[6] 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, US, April 23rd, 2001.

[7] "*ICR C416 - 16-port Dynamic Routing Switch for Transputer Links - Data Sheet*", ICRouting Ltd., 1996.

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

[9] 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.

[10] 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, Aug 2000. pp 334-339.

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

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

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

[14] Liew, E. W. K. *"A Distributed Real-Time Operating System for a Multi-Processor StrongARM Network"*. PhD thesis, The Nottingham Trent University, February 2002.

[15] Wong, K. L. *"A Message Controller For Distributed Processing Systems"*, Ph.D. thesis, The Nottingham Trent University, 2000.

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

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