Massively Distributed Control Nets for Modular Reconfigurable Robots

Ying Zhang, Kimon Roufas, Mark Yim, Craig Eldershaw

Systems and Practices Lab
Palo Alto Research Center
Palo Alto, CA 94304
{yzhang, kroufas, yim, celdersh}@parc.com

Abstract

Massively Distributed Control Nets (MDCN) is a CAN (Controller Area Network) based high-level protocol that has the following features: (1) Three types of communication: individual, group and broadcast, with eight priority levels. (2) Addressing of up to 254 nodes and groups in standard CAN format, and up to 100,000's in extended CAN format. (3) I/O (node-to-node) and port (point/process-to-point/process) communications, where I/O is mostly reserved for system processes with high priorities and short message sizes, and port is for user applications, with lower priorities and possibly large message sizes. Compared to the existing widely used high-level CAN protocols, MDCN can address more communication nodes, has simpler APIs, is easier and more efficient to implement. Also the bridge protocol is transparent to users as whether it is a single CAN bus or a networked CAN buses. MDCN is currently implemented in C for MPC555 TouCAN controller on the Real-Time Operating Systems vxWorks, and in Java on host PC. The API is designed and implemented for multi-threaded environments. MDCN is a general protocol that not only can be applied to modular robots, but also can be applied to any industry control or automation using CAN bus network with hundreds of communication nodes.

1. Introduction

Modular, self-reconfigurable robots show the promise of great versatility, robustness and low cost (Yim, Duff and Roufas, 2000). However, programming such robots for specific tasks, with hundreds of modules and each of which with multiple actuators and sensors, can be tedious and error-prone. The extreme versatility of the modular systems requires a new paradigm in programming (Zhang, Roufas and Yim, 2001). This paper presents the underlying communication structure for a type of modular self-reconfigurable robot, named PolyBot (Figure 1), developed at Palo Alto Research Center [http://www.parc.com/modrobots/PolyBot/polybot.htm].

PolyBot has been designed for applications including planetary exploration, undersea mining, search and rescue and other tasks in unstructured, unknown environments. PolyBot consists of two types of modules: segment modules and node modules. A segment module is composed of two identical connection plates, actuation mechanisms for one degree of freedom rotation, and a Motorola PowerPC *MPC555* embedded processor with 448K internal flash ROM and 1M of external RAM, which also has two internal CAN (Controller Area Network) controllers. The connection plate serves two purposes: to attach two modules physically together as well as electrically; both power and communications are passed from module to module. Two connection plates can be attached and detached via latch actuation, which enables self-reconfiguration. Each connection plate has four IR phototransistors and four IR LEDs. Combinations of IR intensity measurements allow the determination of the relative six degrees of freedom position and orientation of the mating plates, which aids in the closed loop docking of two modules (Roufas et. al, 2001). Each segment module communicates over a global CAN bus with up to 1M bps. A node module is a rigid cube made of six connection plates (one for each face). In addition to a MPC555, there are six external CAN controllers, each for a connection plate. A node module serves three purposes: (1) to allow for non-serial chains/parallel structures, (2) to house higher power computation and power supplies, and (3) to perform transparent media access control (MAC) layer bridging between CAN networks.

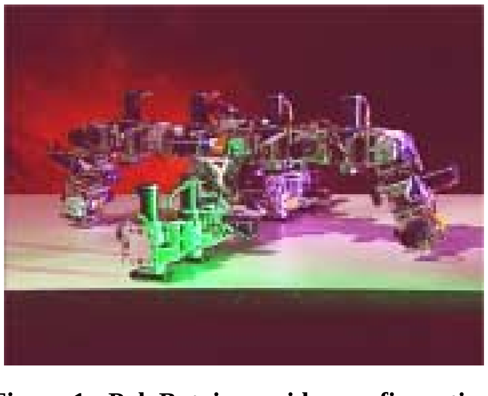

Figure 1: PolyBot in a spider configuration

The PolyBot systems have demonstrated versatility by showing multiple modes of locomotion with a variety of characteristics, distributed manipulation and the ability to self-reconfigure (Yim, Duff and Roufas, 2000)(Roufas et.

al, 2001). For the end of this year, there will be 100 to 200 modules built and connected with various configurations.

Since the electrical design of PolyBot uses MPC555 that has two internal CAN controllers, the use of CAN for communication among modules is convenient. CAN has gained widespread popularity not only in the automotive industry but also in the industrial automation arena (Lawrenz, 1997). CAN has also proven that it fits very well into the suite of field-buses or sensor/actuator buses because of its low price, multiple sources, highly robust performance and already widespread acceptance (Negley, 2000). However, CAN protocol is low level, directly linked to the physical media, which makes the communication programming not only tedious but also ad hoc. Several high-level CAN protocols exist and are widely used, such as CANopen, DeviceNet, SDS and OSEK [http://www.can-cia.de/]. For our application (modular robots), the main limitation of these protocols is their inability to address more than 200 communication nodes. Furthermore, most of these protocols are far more complex and less efficient than our purpose requires.

Massively Distributed Control Nets (MDCN) is a CAN based high-level protocol that has the following features: (1) Three types of communication: individual, group and broadcast, with eight priority levels. (2) Addressing of up to 254 nodes and groups in standard CAN format, and up to 100,000's in extended CAN format. (3) I/O (node-to-node) and port (point/process-to-point/process) communications, where I/O is mostly reserved for system processes with high priorities and short message sizes, and port is for user applications, with lower priorities and possibly large message sizes. Compared to the existing widely used high-level CAN protocols, MDCN can address more communication nodes, has simpler APIs, is easier and more efficient to implement. Also the protocol is transparent to either a single CAN bus or a networked CAN buses. MDCN is currently implemented in C for MPC555 TouCAN controller on the Real-Time Operating Systems vxWorks, and in Java on host PC. The API is designed and implemented for multi-threaded environments. MDCN is a general protocol that not only can be applied to modular robots, but also can be applied to any industry control or automation using CAN bus network with hundreds of communication nodes.

The rest of this paper is organized as follows. Section 2 describes MDCN functionality and APIs. Section 3 presents MDCN protocols. Section 4 illustrates the implementations for embedded processors and the host PC. Section 5 discusses the bridging protocol and implementation. Section 6 concludes the paper.

2. MDCN APIs

Each CAN message provides a standard 11 bits or an extended 29 bits of prioritized destination identification, and eight bytes of data. Priority arbitration, error detection and re-transmission are all handled by the CAN controller hardware (Lawrenz, 1997). However, CAN is low level, directly linked to the physical media, which makes the communication programming not only tedious but also ad hoc. For most applications, a higher-level protocol is necessary. In general, a higher-level protocol handles the following issues:

- *Communication buffers*: ingress and egress queues.
- *Communication configuration*: master/slave, point-to-point, broadcast, group communications.
- *Communication patterns*: block/non-block read/write, confirmation or handshaking, subscribe/publish structures, etc.
- *Fragmentation and reassembly* of large messages.
- *High-level error detection and correction.*

Several high-level CAN protocols exist and are widely used, such as CANopen, DeviceNet, SDS and OSEK (Lawrenz, 1997). For our application, the main limitation of these protocols is their inability to address more than 200 communication nodes, which restricts the scalability of modular reconfigurable systems. Furthermore, most of these protocols are far more complex than our purpose requires.

We have developed a high-level CAN protocol, called Massively Distributed Control Nets (MDCN). Each communication node (simply refer to node in the rest of the text) on a CAN bus has a unique ID, refer as MAC ID, and each node can belong to a set of groups. MDCN features a simple set of APIs, with the following functionalities:

- Three types of communication: individual, group and broadcast, with eight priority levels.
- Addressing of up to 254 nodes and groups in standard CAN format (8 out of 11 ID bits for addresses), and up to 100,000's in extended CAN format (17 out of 29 ID bits for addresses).
- I/O (node-to-node) and port (point/process-to-point/process) communications, where the I/O type is mostly reserved for system processes with high priorities and short message sizes that can be encoded in one data frame, and the port type is for user applications, with lower priorities and possibly large message sizes encoded possibly in many data frames.

The set of MDCN APIs is very similar to those in socket programming, including functions such as *create* and *destroy* port connections, add and remove groups, read/write a message from/to a port. For example, the following code fragment shows that a client is creating a connection and then sending out a message, and the server is accepting the connection and receiving the message.

Client:
```
   //create connection request
   port = createConnection(type, id);
   //write to the connection port
   write(port, message, length, priority);
```
Server:
```
   //accept the connection request
   port = acceptConnection();
```

```
//read from the connection port
read(port, message, &length, timeout);
```

In the above code fragment, type can be INDIVIDUAL, GROUP, or BROADCAST. This allows addressing individual nodes, subsets of nodes (GROUP) or all nodes (BROADCAST). The id parameter is the communication MAC ID for INDIVIDUAL type, or group ID for GROUP type, and read can be blocking or non-blocking depending on the timeout value: -1 means blocking, 0 means non-blocking and any positive number indicates the maximum block time. Before using any MDCN functions except the setMAC_ID function, MDCN init function has to be called, which initializes CAN channels and data structures, as well as starts the MDCN daemon. All the nodes have unique and constant MAC IDs after initialization.

Each port can associate with a *port thread* which co-existent with the port and will be destroyed whenever the port is destroyed. Port threads are mostly used for client/server applications, such that whenever a connection is established between the two parities, the server port runs a daemon for accepting and replying the client's requests.

In addition, MDCN provides special functions for synchronizing clocks and events in a shared bus. The synchronization among nodes is very important for many real-time applications. For example in the PolyBot situation, IR functions on the two opposing plates have to start at the same time to obtain the correct 6D offset reading (Roufas et. al, 2001).

MDCN is also extendable. In addition to create/destroy connections, add/remove groups, which are I/O communications, users can define their own I/O commands. I/O communications are efficient for short (six bytes of data) messages and non-blocking data processing. The difference between port and I/O messages in implementation will be presented in the following sections.

3. MDCN Protocol

Each CAN message provides a standard 11 bits or an extended 29 bits of prioritized destination identification, and eight bytes of data. Priority arbitration, error detection and re-transmission are all handled by the CAN controller hardware.

For standard identification, the 11 bits are assigned as follows in MDCN protocol:

0	1	2	3	4	5	6	7	8	9	10
P0	P1	P2	A0	A1	A2	A3	A4	A5	A6	A7

The first three bits are for priorities, based on the arbitration mechanism built in CAN controllers; so there are eight levels of priorities. The last eight bits are used for MDCN addresses. We divide the whole address space into four categories:

- *Special messages*: address 0
- *Broadcast messages*: address 1
- *Group messages*: address 2 upward
- *Individual messages*: address 255 downward

There are 8 special messages. The receiving of the special messages shall cause the reset of the CAN clock. The one with the highest priority (ID 0x000) is reserved for generating a synchronization signal. Tasks running on different nodes can be synchronized while waiting for that signal, if they share the same bus. Also ID 0x100 is used for registration request for nodes from bridges, and ID 0x200 is used for (BPDU) bridging protocol data units for bridges, --- in our application, bridges are nodes as well, --- the rest of special messages are undefined so far and can be extended later. Broadcast messages are messages that shall be received for all nodes in the network, bridged or not. The individual messages are for individual nodes. Each node is assigned a MAC ID. The MDCN address is (255 – MAC) for individual messages. An individual message shall be received by only one node in the network, assuming all the nodes have different MAC IDs. In addition to broadcast and individual messages, there are group messages. However, the total number of groups plus the total number of nodes cannot exceed 254 for the standard CAN format. For a group with group ID, the MDCN address is (2 + ID). Each node can associate with a set of groups. If a node is associated with a group, the node is a member of that group. A group message shall be received by all members of that group, no matter where the node is in a bridged network.

Each CAN message data frame can have maximum 8 bytes of data; the first two bytes are reserved for MDCN headers. In particular, the first byte is for storing the MAC ID of the source, and the second is for recording the format:

0	1	2	3	4	5	6	7
FF	Tp	Dir	P0	P1	P2	P3	P4

where FF is Fragment Flag. If FF is 1, it is a fragmented message, i.e., message that consists of multiple CAN frames; otherwise, it is a non-fragmented message. If FF is 1, Tp indicates if it is the first frame in a series of fragmented messages, i.e., if Tp is 1, it is the first, and otherwise it is not. For the first fragment frame, three more data bytes are used for recording the length of the total message, i.e., the next two data bytes are reserved for recording the number of frames--the total number of frames can be 2^{16}--and the following byte is for recording the number of bytes in the last frame. On the other hand, if FF is 0, Tp indicates if it is a port or an I/O message. If Tp is 1, it is a port message, and otherwise it is an I/O message. All the fragmented messages are port messages, i.e., I/O messages cannot be fragmented. I/O messages are for system commands or high priority communications. An I/O message is sent to the node(s) it is targeted, a port message is further dispatched to a queue and then accessed by the correspondent port through the read function. The third bit Dir indicates the direction of the communication, i.e., client/request or server/respond. The end that requests for establishing the communication is called *client* and the end that confirms the request is called *server*. A node can have many clients and servers running at the same time. The last 5 bits indicate the client port ID of the communication. Therefore, there are maximum 32 client ports that can be active at the same time in any node.

When the extended CAN format is used, the 29 bits ID is assigned as follows: the first three bits are still for priorities; there are 8 levels of priorities, the last 17 bits are for MDCN addresses, and the middle 9 bits are for the first 9 bits of the MAC ID source. There are also 8 special messages with MDCN address 0, and two data bytes reserved: the first byte is the last 8 bits of MAC ID source and the second byte stores the same information as the standard format. The extended format can address up to (2^{17} – 2), i.e., more than 100,000, groups and individual nodes.

4. MDCN Implementation

MDCN has been implemented on the embedded processors MPC555 in C using RTOS vxWorks, and on the host PC in Java. In the former case, IEEE standard POSIX pthreads are used, so that the code can be migrated to other operating systems easily. The implementation enables the communication between the host PC and the target, as well as the communication among target processors in a uniform format. CAN drivers are implemented differently for embedded chips and for the host PC, but the high level MDCN implementations share the same structure.

4.1 CAN Drivers

4.1.1 MPC555 TouCAN Driver

A modified version of CANpie [http://www.microcontrol.net/CANpie/index.html], mCANpie, is implemented for MPC555 TouCAN driver. For a node with one CAN controller (channel), an input queue and an output queue are initialized. In addition, buffers are allocated: one for receiving special messages (address 00), one for sending special messages, one for receiving individual messages, one for receiving broadcast messages (address 01), and one for sending regular I/O or port messages. Buffers for receiving group messages will be allocated when the node is added to the group, and be deallocated when the node is removed from the group. For MPC555 TouCAN, each node can be associated with at most 11 groups since there are total 16 buffers, 5 of which are already used. The multi-buffer makes the implementation efficient since the filters are hardware instead of software; each node will only receive the messages sent to it. For a bridge, multi-buffer is not necessary since it should receive all the messages sent on the bus.

The input and output queues are implemented to be thread-safe, so that MDCN APIs works in multi-threaded environments. Also, the CAN driver is interrupt driven rather than frequent pulling. The receiver handler and the transmit handler will be triggered whenever an input comes in or an output is sent out, in the interrupt routine. The transmit handler would simply take out another data frame from the output queue and send it out. The receiver handler for a node is simple, which pushes any non-special messages to the input queue. For special messages, it will call the correspondent routines accordingly, such as triggering a synchronization mechanism, or sending out a register message to bridges. The receiving handler for bridges will be discussed in Section 5.

4.1.2 Host PC CANCardX Driver

For PC, the CANalyzer from Vector Informatic GmbH [http://www.vector-informatik.de/] is used. A JINI interface is implemented to call read and writer functions from the CANDriver. The host PC does not process special messages, nor it belongs to any group. Like any node, the host has a MAC ID. It receives individual and broadcast messages only, even though it can still sends group messages.

4.2 MDCN Ports and FIFOs

The main data structure for MDCN is a port, which acts as a connector to the other end of the communication. The maximum number of ports is predefined and allocated in an array, so that ports can be accessed through the index in the array. The maximum number of client ports is limited by 32 (5 bits). A port data structure is defined as follows:

```
typedef struct {

  _BIT        status;      //TURE: in use; FALSE; in idle
  _U08         portID;     //ID of the client port
  _BIT         direction;  //client or server
  _U08         type;       //broadcast, individual or group
  MAC_ID   macID;         //8 or 17 bit ID of the destination
  MDCN_FIFO *fifo;     //its receive FIFO queue
  pthread_t thread;    //thread running on the port

} MDCN_Port;
```

where MDCN_FIFO is a queue that receives messages directing to that port. In the case of broadcast or group communication, a client can have many servers. In order to receive replies with large data sizes composed of many fragments from many servers, MDCN_FIFO is defined as a list of FIFO queues, so that a port can get replies from many sources, each of which is directed to one queue. The mechanisms for selecting the right FIFO for push and pop functions are implemented.

The MDCN_FIFO is defined as follows:

```
typedef struct MDCN_FIFO_STRUCT {
   pthread_mutex_t  mutex; //multi-thread protection
   pthread_cond_t   cond;   //conditional variable
   DATA             queue[FIFO_SIZE]; //FIFO queue
   _U16             head; //head of the queue
   _U16             tail; //tail of the queue
   _U16                   size; //number of elements
   MAC_ID                 source; //source of reply
   struct MDCN_FIFO_STRUCT  *next; //linked queue
} MDCN_FIFO;
```

where mutex is for the multi-threaded access protection, and cond is for conditional blocking when the queue is empty for pop or when the queue is full for push. The MDCN write operation will put the fragmented data frames directly in the CANpie output queue, by recording source in the first byte, the direction and port ID of the port in the second byte, for each data frame. The MDCN read operation gets a set of data frames from the first non-empty FIFO queue of the port, and assembles the fragmented data frames to a message if necessary. If the port is client and its type is group or broadcast, the user has the responsibility for reading all the messages from all the sources by calling the read function as many times as necessary; use the timeout read to make sure all replies are read.

4.3 MDCN Initialization and Daemon

Each node for MDCN has to be initialized before calling any MDCN functions except the function for set MAC ID, which has to be done before the initialization. Initialization process initializes communication channels and data structures, sets up receiving message buffers for special, individual and broadcast messages, finally starts the MDCN gateway daemon, the flowchart for the function of the daemon is shown in Figure 2.

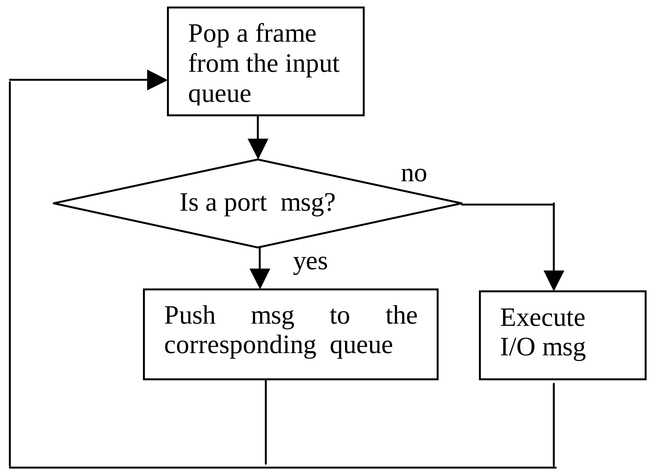
Pop a frame from the input queue
Is a port msg?
no
yes
Push msg to the corresponding queue
Execute I/O msg

Figure 2: MDCN gateway daemon

If the message is from a client, then portID and source will be used for finding the corresponding port, otherwise if it is from a server, portID is the index of the port. Also, if the message is from a server and the type of the port is group or broadcast, and if that message is fragmented, the message will be further de-multiplexed to the MDCN_FIFO of a single source. In this implementation, I/O messages are processed directly in MDCN daemons; port messages are dispatched to corresponding queues, and processed by separate threads/processes.

There are four I/O messages implemented so far, create and destroy connections, add and remove groups. The request for create connection from a client will be pushed to the connection request queue, which will be accessed by AcceptConnection() called from the server. The requests for destroy connection, add or remove groups are processed directly in the daemon loop; destroy connection will remove the active server port, add group will record the group and also allocate a message buffer from the CAN controller for receiving group messages, and remove group will remove the group from the list and also de-allocate the message buffer.

5. MDCN Bridging

Even though MDCN can address up to 100,000 nodes in the extended CAN format, CAN has the limitation for the maximum numbers of nodes in any single bus. Bridges are necessary for a system to be scalable. In general, a bridge is a node with more than one channel; any two channels from the same or different nodes can be connected. A bridge has a routing table and forwards messages from one bus to another. In PolyBot application, a node module acts as a bridge; it has six channels. A communication network with bridges consists of more than one bus. MDCN APIs are transparent on whether there is a single or bridged CANbus network. An MDCN bridge is also an MDCN node. The MDCN bridge is designed and implemented based on the ANSI/IEEE Standard on “Media Access Control (MAC) Bridges”, 802.1D. Since most part of the bridge code is standard, we will only describe the connection between MDCN nodes and MDCN bridges.

The initialization process for the bridges also starts the MDCN bridge daemon, in addition to the MDCN daemon, since in general, bridge itself also acts as a communication node. There are n output queues for n channels, but only one input queue, which receives incoming non-special messages from all channels targeted to that node. In addition, there is a bridge messages queue for the bridge daemon. The receive handler for the bridge called from the interrupt routine dispatches the messages according to the IDs; it first filters out and processes the special messages. There are three special messages: 0x000 for synchronization, 0x100 for registration request for all MDCN nodes, and 0x200 for bridge related information that shall be processed by bridges only. The synchronization message will generate a synchronization event; a registration request message will trigger a registration message to send back through the receiving channel. All bridge related information received by bridges are pushed into the bridge queue and processed by the bridge daemon. For a non-special message, if the receiving channel is active, it either pushes the message to the input queue that shall be processed by the MDCN daemon and/or forward the message to other active channels. Figure 3 shows the flowchart for the bridge receive handler.

The bridge daemon handles all the messages to the bridge queue. There are three types of messages: configuration message CONFIG_BPDU_TYPE, topology change notice message TCN_BPDU_TYPE and register message REGISTER_TYPE. The first two types of messages are handled the same way as the IEEE standard, the last type is for updating the forwarding table. The request for register can be triggered every time the channel becomes enabled, and/or every fixed sampling time for verification if necessary, by setting the time interval. There are two types of registry: local and global. Local registry only updates the forwarding table in the current bridge and global registry propagates the request to all the other enabled channels.

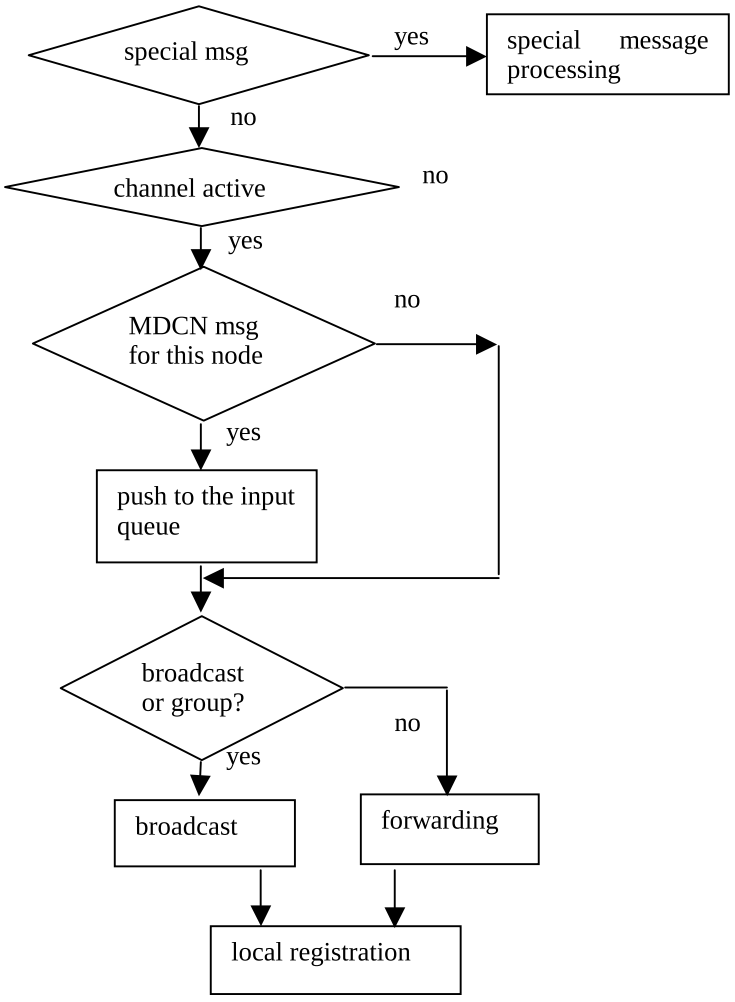

Figure 3. Bridge Receiving Handler

We should point out that in the case of bridged networks, messages across the bridges will be delayed. In the case of broadcast or group communication in a bridged network, nodes in different buses receive the same message at different times.

6. Conclusions

We have presented here MDCN: Massively Distributed Control Nets as a higher level protocol for CANbus based communications, and its implementation for MPC555 TouCAN controllers and the host PC. MDCN supports three types of communication: individual, group and broadcast, and can address up to 254 nodes in standard CAN format and 100,000 nodes in extended CAN format. With the integration of bridging, it is totally transparent from users point of view if it is a one bus or a connected set of buses. MDCN has been used for implementing Attribute/Service Model (Zhang, Roufas and Yim, 2001)(Zhang et. al, 2002), with CAN as the communication medium, for modular self-reconfiguable robots. However, MDCN is a general protocol that can be applied not only to the modular self-reconfigurable robots but also to any industry control platform using CANbus.

References

[1] W. Lawrenz, CAN System Engineering: From Theory to Practical Applications, Springer, 1997.

[2] M. Yim, D. Duff, K. Roufas, “PolyBot: a Modular Reconfigurable Robot” *Proc. of the IEEE Int. Conf. on Robotics and Automation*, April 2000.

[3] B. Negley, “Getting Control Through CAN,” *Sensors*, vol. 17, no. 10, pp18-34, October 2000, also available in http://www.sensorsmag.com/.

[4] K. Roufas, Y. Zhang, D. Duff, M. Yim, “Six Degree of Freedom Sensing for Docking using IR LED Emitters and Receivers,” *Experimental Robotics VII*, Lecture Notes in Control and Information Sciences 271, D. Rus and S. Singh Eds. Springer, 2001.

[5] Y. Zhang, K. Roufas, M. Yim, “Software Architecture for Self-Reconfiguable Robots”, Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, October, 2001.

[6] Y. Zhang, M. Yim, K. Roufas, C. Eldershaw and D. Duff, “Attribute/Service Model: Design Patterns for Distributed Coordination of Actuators, Sensors and Tasks”, submitted, 2002.

Acknowledgement

This work is funded in part by the Defense Advanced Research Project Agency (DARPA) contract # MDA972-98-C-0009.