RESEARCHERID
THOMSON REUTERS

[Dülger* 6(4): April, 2019] ISSN 2349-4506
Impact Factor: 3.799

Global Journal of Engineering Science and Research Management

VIRTUAL MODEL FOR THE SIMULATION OF THE CONTROLLER AREA NETWORK

Mustafa Dülger
Mechanical Engineering Department, Faculty of Engineering, University of Istanbul Cerrahpasa, Istanbul 34320, Turkey

DOI: 10.5281/zenodo.2656480

KEYWORDS: Distributed Control System, Controller Area Network, Simulation, C++, Virtual Model.

ABSTRACT

In this work, a virtual model for the simulation of *Controller Area Network (CAN)* transmission protocol is proposed. The proposed model is named as the *Can Bus Simulation Model (CBSM).* Underlying hardware of the *CAN* protocol is ignored as the objective is to lay down a base model for the grand process which is to be simulated under the *windows* operating system. The model is composed of virtual *nodes*, virtual *bus* and virtual *container* objects. The node *objects* interchange CAN message over the *bus* object. The *bus* object acts as if it were a CAN controller. The container object is the place holder for the bus and nodes objects. The model is based on the *COM (Component Object Model)* technology. A dynamic link library, *CanServer.dll,* implementing the proposed model is developed in C++. The library is made available by the author if requested

INTRODUCTION

The central control system is losing importance as the trend is towards decentralized embedded control system. Many engineering problems are now modelled by the decentralized embedded models. The rapid development in the electronics of field protocols has accelerated this trend. Parallel to this trend, simulation of field protocols helps simulation of embedded models and speeds up the development process.

One of the remarkable characteristic of the embedded control system is the full abstraction and separation of the communication part from the main process. This let the process developer fully focus on the process and not on the communication. Another important issue is the simulation of the process on computer to accelerate the process development. The work presented in this manuscript reveals a virtual model for the simulation of the *Controller Area Network* (*CAN*) communication protocol [1]. The proposed model is named as the *Can Bus Simulation Model* (*CBSM).* The goal is to automate can communication protocol under windows environment and develop a dynamic link library for the simulation. When a process developer needs to simulate his own process under *windows* operating system, he will not have to deal with the communication issues among the process objects. Instead the library will take over the communication tasks. He needs only link the library to the grand process. *CBSM* model is based on the *Component Object Modelling (COM)* technology [2]. A dynamic link library, *CanServer.dll,* implementing the proposed model, is developed in C++. The library is made available by the author if it is requested.

CONTROLLER AREA METWORK MODEL

Fig.1 illustrates the *Controller Area Network Model* (*CANM*) for a distributed system. The *CANM* has *m node* objects being connected over the can bus protocol. Each node should interact with the process *object* through the well-defined process interface. Design of the process object is the duty of the process designer.

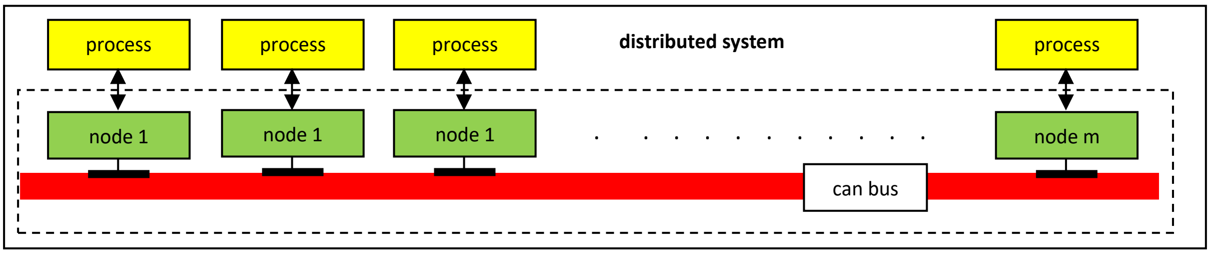

Fig.1: Controller Area Network Model for a distributed system having m nodes.

CAN BUS SIMULATION MODEL (CBSM)

Can Bus Simulation Model, *CBSM*, is the virtual model designed in order to simulate *Controller Area Network Model*, *CANM*, introduced in the previous section. Fig.2 shows interactions diagram of the objects in *CBSM*. *CBSM* has two classes. They are *CCanBus* and *CCanNode* classes. Only one object of the *CCanBus* class is created and named as the *theBus* object. The name convention of the *node* objects is as follows. *node-1 is* for the first object, *node-2 is* for the second object and *node-m* for the m th object and so forth.

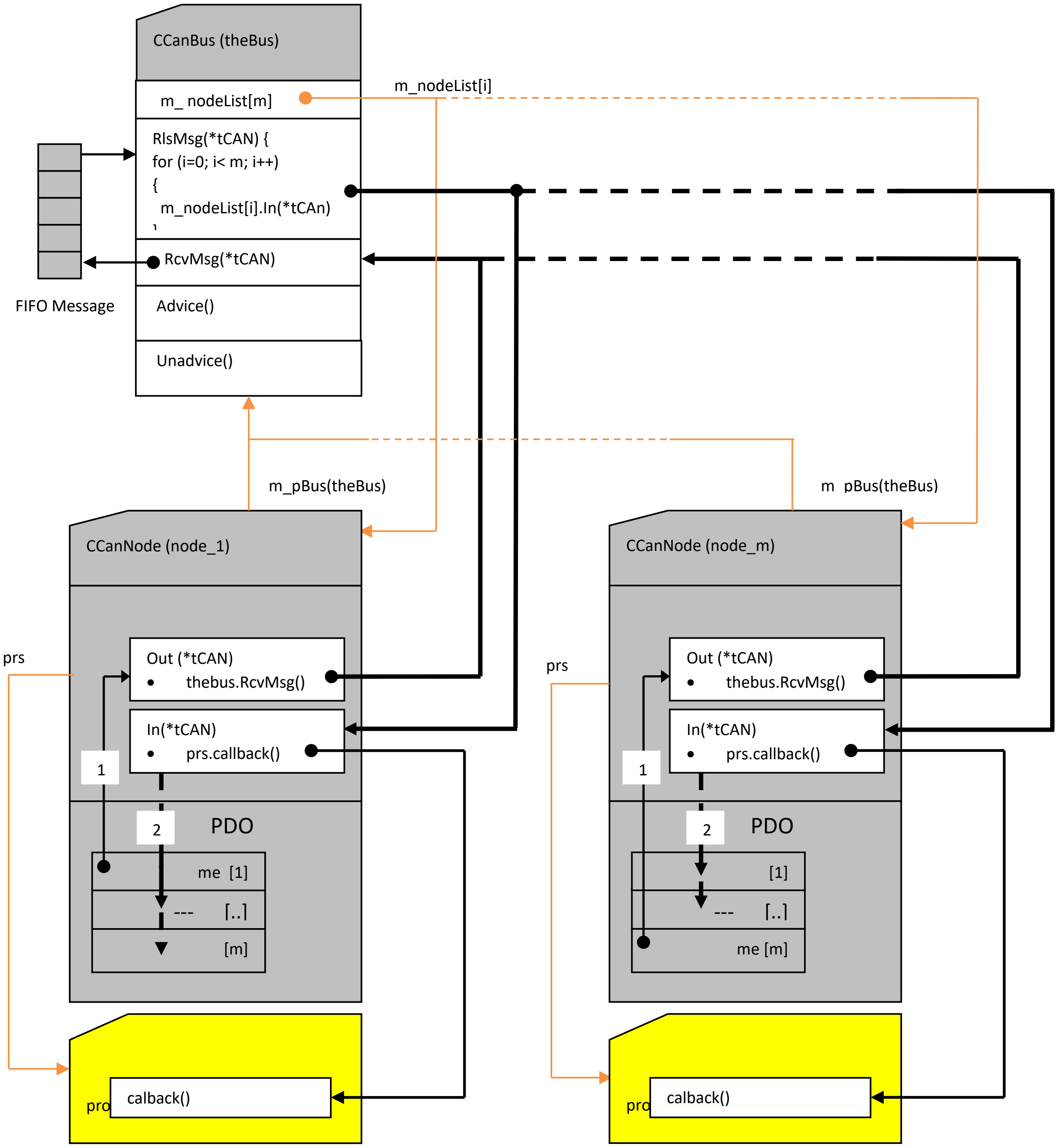

Fig.2: Interaction diagram of the objects in Can Bus Simulation Model, CBSM.

Operation of Can Bus Simulation Model

As it is seen from Fig.2, at least two *node* objects and a single bus object, *theBus* object, are created for a minimum system. All *node* objects are preregistered to the *theBus* object. Registration takes places by calling the *advice(IUnknown** ppNode, LONG* cookie)* method of the *theBus* object. The address of the pointer to the *node* object is passed as an argument. The passed *node* pointer is then stored in an internal list named as *m_NodeList[]*. A cookie for the registered *node* object is returned to the calling function. The cookie is later used by the *unAdvice(LONG cookie)* method to unregister the *node* object.

The *node* object which wants to transmit a can message, initiates the transmission by calling the *RcvMsg(*tCAN)* member function of the *theBus* object in it's *Out(*tCAN)* method. The *RcvMsg(*tCAN)* method passes the address of a variable of type *tCAN* as a can message argument and stores the value of the variable in an internal *first-in first-out* (*FIFO*) *Que*.

The *theBus* object continuously observes the *Que*. If there is a non-transmitted can message already sitting in the *Queue*, it is picked up and transmitted by calling the *RlsMsg(*tCAN)* method. The *RlsMsg(*tCAN)* method enumerates all registered *node* objects and passes the can massage by calling *In(*tCAN)* member function of registered *node* objects except the sending one.

The *node* object stores all process variables, encoded in type of *tCAN*, in a local memory block named *Process Data Object*, *PDO*. *PDO* is well structured and incorporates so many sub sections as the number of all registered *node* objects. Each sub section is assigned to the related *node* object. Assignment is accomplished according to the simple indexing. Each *node* object has a unique identification number. The first *node* object has an identification number one, the second two and so forth. The first sub section in *PDO* is assigned to the *node* object having the identification number one, the second sub section in *PDO* is assigned to the *node* object having the identification number two and so forth till the last sub section assigned to the *node* object having the identification number *m* where *m* is the identification number of the last *node* object. When a *node* object must transmit a can message, it creates a function argument of type *tCAN* from the data in the local section of the *PDO* object (me section and path -1- in Fig.2). The process object is responsible for keeping the process parameters up to date. The function argument so created is then passed to the *theBus* object.

When a *node* object receives a *can message*, the incoming function argument of type *tCAN* is first decoded for the sending *node* object identification. The can message is then stored into the related section (path -2- in the Fig.2). The process *callback()* function is then called so that the process is informed that the local *PDO* is updated by another *node* object. At this instant, the process must act in accordance with the changing *PDO* object (process update).

CAN Data Frame Structure tCAN

```
// CanBusServer.idl : IDL source for CanBusServer
// CAN message
typedef
[
      uuid(B268A2C4-2A2E-4c9d-804D-0B2BAB4E47C8)
]
struct tCAN
{
      unsigned short id;   // ID of the can massage (11 Bit)
      int rtr;             // (1)! Remote-Transmit-Request-Frame?
      BYTE length;         // number of data bytes
      BYTE data[8];        // buffer for 8 data bytes of can message
}tCAN;
```

Fig.3: Definition of the Structure tCAN in the CBSM model.

RESEARCHERID
THOMSON REUTERS
[Dülger* 6(4): April, 2019] ISSN 2349-4506
Impact Factor: 3.799
Global Journal of Engineering Science and Research Management

Definition of the structure *tCAN* in *CBSM* is given in Fig.3. Only four fields from the standard CAN Data Frame are used [3]. They are namely can message identification *id*, remote transmit request *rtr*, number of data bytes *length* and buffer for data bytes *data[8]*. Type and size of these parameters are similar to that of defined ones in a standard CAN data frame.

Class CCanBus

CCanBus class is designed for the bus object in the *CBSM*. The header file of the class is given in Fig.4.

```
class ATL_NO_VTABLE CCAN_Bus :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCAN_Bus, &CLSID_CAN_Bus>,
       public IDispatchImpl<ICAN_Bus, &IID_ICAN_Bus, &LIBID_CanBusServerLib,
       /*wMajor =*/ 1, /*wMinor =*/ 0>
{
       public:
       CCAN_Bus();
       DECLARE_REGISTRY_RESOURCEID(IDR_CAN_BUS)
       BEGIN_COM_MAP(CCAN_Bus)
       COM_INTERFACE_ENTRY(ICAN_Bus)
       COM_INTERFACE_ENTRY(IDispatch)
       END_COM_MAP()
       CComDynamicUnkArray m_vecCallBk;          // nodeList
       DECLARE_PROTECT_FINAL_CONSTRUCT()
       public:
       STDMETHOD(Advice)(IUnknown** ppNode, LONG* pCookie);
       STDMETHOD(Unadvice)(LONG cookie);
       STDMETHOD(RlsMsg)();
       STDMETHOD(RcvMsg)(tCAN * pMsg);
       protected:
       std::queue <CCANMsg * >  m_lstFIFO;       // FIFO Queue
       int _RlsMsg();
};
OBJECT_ENTRY_AUTO(__uuidof(CAN_Bus), CCAN_Bus)
```

Fig.4: Header File for the class CCanBus in the CBSM model

CCanBus class implements the public interface, ICAN_Bus having four methods. They are,

- *Advice (IUnknown** ppNode, LONG* pCookie)*
  The method registers the node object and returns a cookie. Address of a pointer to the node object, ppNode, is passed as an argument. A pointer to a LONG parameter, pCookie, is also passed. The parameter pCookie is used as a handle to the registration.
- *Unadvice (LONG cookie)*
  The method unregisters the *node* object. The *node* object is early registered and handled by the parameter cookie.
- *RlsMsg()*
  The method picks up the early can message sitting in the message queue and passes it to the registered *node* objects except the sending *node* object.
- *RcvMsg(tCAN * pMsg)*
  The method receives a can message from a *node* object and places it in the message queue.

Class CCanNode

CCanCNode class is designed for the node object in the *CBSM*. The header file of the class is given in Fig.5. *CCanNode* class implements the public interface, ICAN_Node having four methods. They are,

- *In(tCAN* canMsg);*
  The method is called by the *theBus* object when the bus has a message. The address of a message is passed as an argument. The method calls a callback function which is already registered.
- *Out(tCAN* canOut);*
  The method is called when the *node* object itself wants to transmit a message. The message is given to the *theBus* object.
- *AddCallBack(long** fp, long ** obj);*
  The method registers a callback function pointer, *fp*, over a static object indicated by *obj*. The callback function is implemented by the *process* object.
- *w_on ();*
  The method is reserved for process implementation.

```
class ATL_NO_VTABLE CCAN_Node :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCAN_Node, &CLSID_CAN_Node>,
public IDispatchImpl<ICAN_Node, &IID_ICAN_Node, &LIBID_CanBusServerLib, /*wMajor
=*/ 1, /*wMinor =*/ 0>
{
protected:
       BYTE m_PDO[NUM_NODES][8];         // Process Data Object
       BYTE m_COMMAND[8];
       BYTE m_nMe;
       FP   m_pf;
       void*     m_pOBJ;
       ICAN_Bus* m_pICanBus;             // smart pointer to the theBus
       public:
       CCAN_Node();
       DECLARE_REGISTRY_RESOURCEID(IDR_CAN_NODE)
       BEGIN_COM_MAP(CCAN_Node)
       COM_INTERFACE_ENTRY(ICAN_Node)
       COM_INTERFACE_ENTRY(IDispatch)
       END_COM_MAP()
       DECLARE_PROTECT_FINAL_CONSTRUCT()
       public:
       /* interface ICAN_Node - methods */
       STDMETHOD(In)(tCAN* canMsg);
       STDMETHOD(Out)(tCAN* canOut);
       STDMETHOD(AddCallBack)(long** fp, long ** obj);
       STDMETHOD(w_on)();
       /* interface ICAN_Node - properties */
       STDMETHOD(get_CanBus)(IUnknown** pVal);
       STDMETHOD(put_CanBus)(IUnknown* newVal);
       STDMETHOD(get_me)(BYTE* pVal);
       STDMETHOD(put_me)(BYTE newVal);
};
OBJECT_ENTRY_AUTO(__uuidof(CAN_Node), CCAN_Node)
```

Fig.5: Header File for the class CCanNode in the CBSM model.

Global Journal of Engineering Science and Research Management

TESTING THE CAN BUS SIMULATION MODEL

The developed dynamic link library *CanServer.dll*, implementing the *Can Bus Simulation Model*, *CBSM*, is used and verified in the *Distributed Elevator Control System*, *DECS*, carried out by the author [4]. *DECS* is based upon the *CBSM*. *DECS* is made up of *Lift-Objects* and *Stair-Object* each of which aggregates a *CCanNode (node)* object. The intelligence is distributed among the *Lift-Objects*. All objects are connected over the *Controller Area Network* (*CAN*) through the simulated *CCanBus (theBus)* object. Communication task is hence performed by the *Can Bus Simulation Model*, *CBSM*.

The test model has one *theBus* object, four *Lift-Objects* objects and 10 *Stair-Objects* objects. Each *Lift-Object* should serve the demands coming either from inside the cabin and/or from the stair objects (building). As the *Lift-Objects* are serving the demands, they are either moving up, down or at rest. That means state parameters of the each lift object (lift identification number, current stair at which the lift is passing or staying, direction of the lift movement, demand vector inside the lift cabin etc.) are changing. Similarly state parameters of the each stair object are also changing as the grand process goes forth. How lifts make decision to move is a matter of grand process and discussed in *DECS* in details [4].

Each object must be aware of any change in other object's states. In our test case, each lift and stair object must know state of all other objects simultaneously. Therefore each object broadcasts its state to all other *objects* in a regular time interval continuously thorough the *CBSM*. Here each lift and stair object redraws itself each time after the broadcasting is completed. Continuous redrawing of objects on the screen builds the simulation of the grand process.

Fig.6 is taken from the test. Careful examination of the Fig.6 reveals that objects states are changing as the grand process dictates. This indicates and proofs that the *DECS* works as expected. Since *DECS* utilises the *CBSM* for communication, this indication is also a proof for the reliable operation of the *CBSM*.

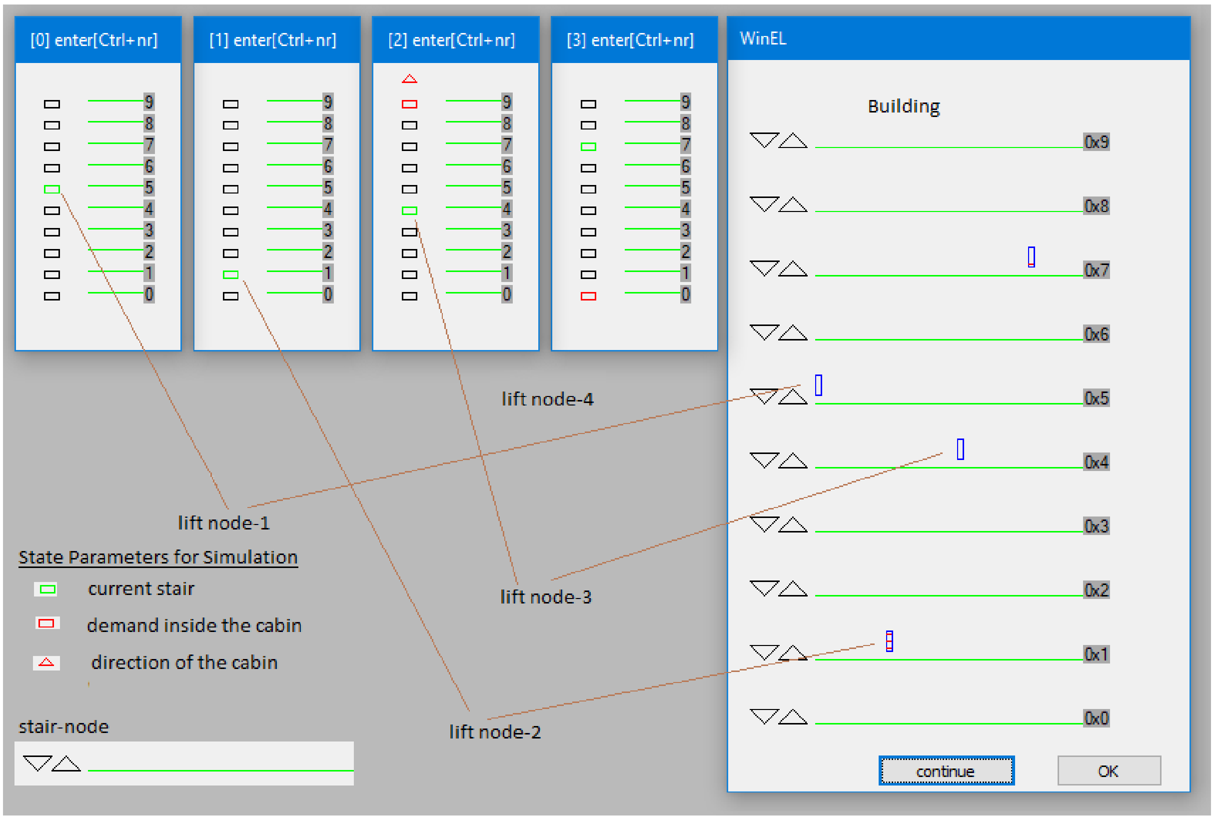

Fig.6: Simulation of the Lift-Objects and Stair-Objects in test case [4].

PROPOSAL FOR FUTURE WORK

In this work, a virtual model for the simulation of the can bus protocol is proposed for the process which is to be simulated under *windows* operating system. Underlying physical and low level specification of the *CAN* protocol is ignored. As a future work, the ignored specification can be implemented. This will help linking the simulations to real processes hence providing more flexibility in design and development of the distributed system.

ABBREVIATIONS

CCS:	*Centralised Control System.*
CAN:	*Controller Area Network.*
COM:	*Compound Object Modelling*
CANM:	*Controller Area Network Model.*
CBSM:	*Can Bus Simulation Model.*
DECS:	*Distributed Elevator Control System.*
tCAN:	*Can Data Frame structure.*
PDO:	*Process Data Object*

REFERENCES

1. Robert Bosch GmbH, “CAN Specification 2.0”, 1991.
2. Microsoft, “The Component Object Model”, https://docs.microsoft.com/en-us/windows/desktop/com/the-component-object-model
3. Cook J.A., Freudenberg J. S., "Controller Area Network”, EECS 461, Fall 2008, Revised October 2013.
4. Dülger, Mustafa “Development of the State Machine for the Distributed Elevator Control System Implementing Control Area Network (CAN)”, Global Journal of Engineering Science and Research Management, February 2019.