DIME: A SHARED MEMORY ENVIRONMENT FOR DISTRIBUTED SIMULATION, MONITORING AND CONTROL OF URBAN TRAFFIC

A. Argile[1], E. Peytchev[1], A. Bargiela[1], I. Kosonen[2]

[1] Real Time Telemetry Systems, Department of Computing,
The Nottingham Trent University, Burton Street, Nottingham, NG1 4BU, UK

[2] Laboratory of Transportation Engineering,
Helsinki University of Technology, Rakentajanaukio 4 A, FIN-02150 Espoo, Finland

Keywords: Shared Memory, Distributed Simulation, TCP/IP, Transportation.

1. Abstract

This paper describes a distributed computers shared memory system which is intended to provide a generic processing harness for the execution of software modules that extend the functionality of current urban traffic control systems. The harness makes use of TCP/IP communication libraries and the design of the system reflects features specific to the traffic control. It is designed to support applications for both DOS and UNIX platforms.

2. Introduction.

The prospect of the enhancement of the performance of current traffic control systems, through the provision of a supervisory level of control using in-vehicle and road-side dynamic route guidance, has created the need for a flexible computing environment in which various new applications can be fully integrated with an existing system without adversely affecting its performance. The most promising approach to satisfying this requirement is the use of distributed computing resources. The distributed computers shared memory system (DCSM) described in this paper, provides a processing environment for the execution of software modules of urban traffic control systems. An implementation of a DCSM system, developed by the authors, has been called DIME which stands for a DIstrubuted Memory Environment.

The purpose of a DCSM system is to allow computational tasks to assume a globally shared virtual memory even though the tasks execute on nodes that do not physically share memory. Figure 1 illustrates the concept of a DCSM system. In this scheme each processor can access global data without the application program having to specify explicitly where this data is or how to obtain it since any request to access shared memory is routed by default to the Memory Manager Processor. This is in sharp contrast to message passing systems where each application needs to take care explicitly of interprocess communication, which is a potentially complex and error prone task. An additional advantage of DCSM is that it provides the same programming environment as hardware shared memory multiprocessors. Programs written for a DCSM system are easily ported to a shared memory multiprocessor. However, porting from a multiprocessor system to a DCSM system may require more modifications because the DCSM's higher latencies put a greater value on locality of memory access.

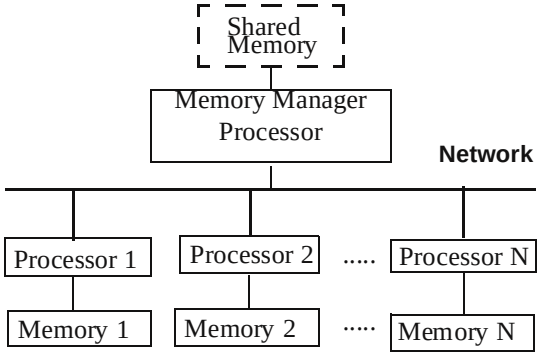

Figure 1. Distributed computers shared memory (DCSM) model

Two important concerns that characterize DCSM systems are *memory structure* and *memory consistency model*. The shared memory may be provided as an unstructured linear array of bytes or it may be structured and organized in terms of objects like lists, circular buffers, records etc. The consistency model refers to how shared memory system updates become visible to other processes. The intuitive model is that read should always return "the last value written". Unfortunately, the notion of "last value written" is not well defined in distributed systems. A more precise notion is *sequential consistency*, in which all processes see memory as if they were running on a single processor [4]. With *sequential consistency* the "last value written" is defined precisely, and although sequencing the access to memory implies deterioration of performance, the shared memory systems based on sequential access are quite adequate for a large class of applications.

3. Design

Our implementation of a general DCSM model, the DIME, provides structured shared memory and it uses sequential consistency model implemented as a centralized memory manager. This design implies that DIME does not require any virtual memory

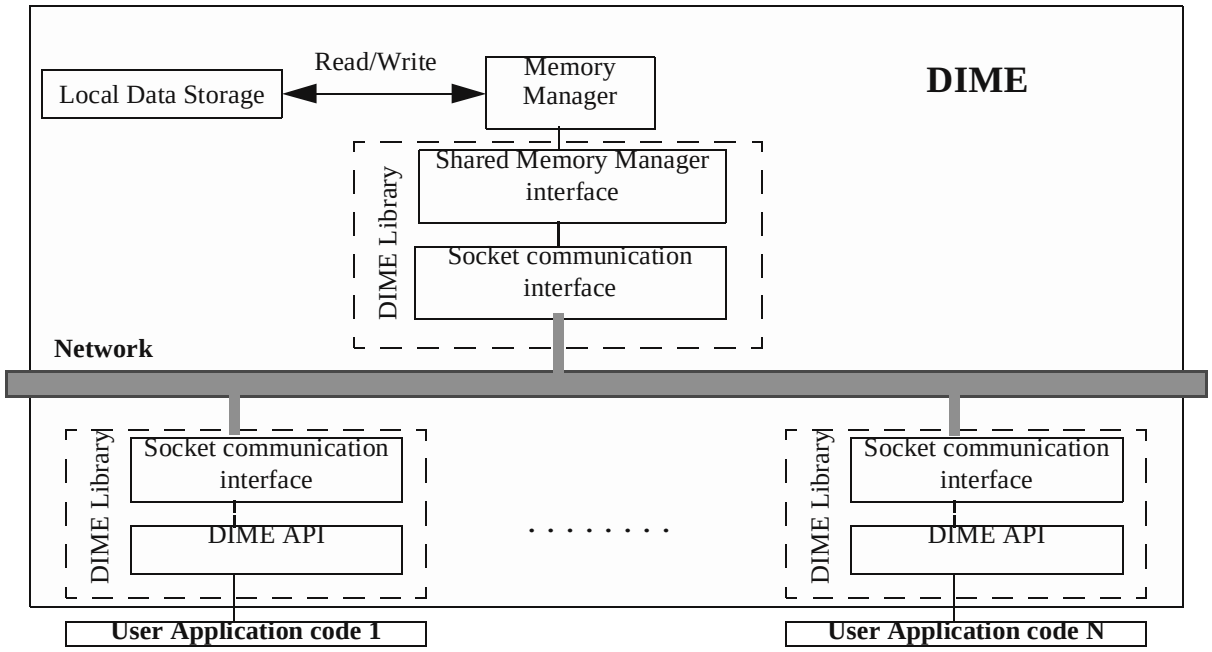

Figure 2. DIME configuration

hardware, to detect accesses to pages of shared memory, and consequently it could be ported to both Unix and DOS environments. DIME system is implemented entirely as a user-level library on top of UNIX and DOS. Unix kernel modifications were unnecessary because all required communication and memory management functions are standard features of current Unix implementations. The DOS implementation of DIME makes use of socket communication libraries from 3COM, [1], which provide full compatibility with the Unix implementation. Programs written in C or C++ are compiled and linked with DIME library using any standard compiler for that language. As a result, the system is relatively portable. DIME's configuration is represented on Figure 2. Each user application code has additional component linked to the code, which provides the communication interface via DIME API with the shared memory system. The requests for reading/writing data from/to the shared memory (creating or removing areas) are transferred by the DIME library over the network to the memory manager task, where they are being processed and replies are sent back. There are two components of DIME: a) shared memory manager (SMM) task which owns the shared area and b) communication DIME libraries which are linked to user applications and the memory manager in order to interface to the network.

The shared memory manager (SMM) component of DIME operates on a closed-loop basis, continually checking for the requests to access or to maintain the shared memory data structures. These requests are typically raised by the application programs executing appropriate API routines, but a provision has also been made for a keyboard entry of requests to the memory manager.

The DIME's API provides facilities for creation and removal of shared memory structures and the read/ write access to them. The synchronization of accesses to shared memory is implicit in the operation of the memory manager task thus it does not require the provision of separate library functions.

4. Shared Memory Categories in DIME.

In supporting the traffic simulation, monitoring and control applications, DIME software implements two types of memory structures, an array of records and a circular buffer. These two structures are intended for use with static and dynamic data respectively. Typically, an array of records will store a network description data, that is shared between several concurrent applications and is read and updated only infrequently. Because of that, it is possible to afford a simple model for accessing the shared arrays whereby always a complete array is read or written-to. For the time-varying data, such as the traffic flow measurements or the results produced by the real-time simulation and control software modules, the access to the supporting memory structures is requested by applications on a second-by-second basis, so the efficiency of access to the shared memory is an utmost priority. The circular buffer data structure provides an efficient access to an individual record and it readily keeps track of the time sequence of messages. The buffer maintains a global insertion pointer and an individual extraction

pointers for each of the "readers", thus enabling applications to recover from sporadic communication delays that are inevitable in a distributed processing environment.

5. DIME's API

The C language interface is as follows:

/* initialize shared memory API interface */
Boolean init_DSM (int argc, char **argv, int MaxAreas, char *host, int port);
Application program supplies "C" standard command line parameters **argc** (command line argument count) and **argv** (command line arguments represented as array of strings) to API. The start-up options include: a) specifying the host name (or address) of the computer where the Memory Manager Task is running; b) specifying blocking or non-blocking mode for socket operations. The name of the Memory Manager Task's host could be supplied by using the fourth parameter of the function **host**, in which case there is no need to specify it in the start-up parameters. The parameter **MaxAreas** determines the maximum number of shared areas for this specific application and **port** is a parameter required for establishing TCP/IP socket communication interface between the application tasks and the memory manager task (usually predetermined by the DIME system).

/* create shared memory array */
Boolean request_create_array (String name, int recordsize, long records, char *taskname, int permission);
Application program specifies parameters for the name of the array, the size of one record, the number of records in the array, the name of the requesting task and the type of access required.
The request by the application program to create shared array that already exists has an effect of checking the memory manager's specification for this array against the applications' specification and if the two agree the memory manager authorizes the application to perform subsequent read/write operations.

/* remove shared memory array */
Boolean request_remove_array (String name, char *taskname,);
Application program specifies parameters for the name of the array and the name of the requesting task. The request results in the name of the application being removed from the memory manager's list of tasks that are permitted to access the array. If the requesting task is the last on the list, the shared array is removed.

/* read from shared memory array */
Boolean request_read_array (String name, Address tolocaladdress);
Application program specifies parameters for the name of the array and the address where data go to.

/* write into shared memory array */
Boolean request_write_array (String name, Address fromlocaladdress);
Application program specifies parameters for the name of the array and the address where data comes from.

/* create a circular buffer */
Boolean request_create_buffer (String name, int recordsize, long records, char *taskname, int permission)
Application program specifies parameters for the name of the buffer, the size of one record in the buffer, the maximum number of records in the buffer, the name of the requesting task and the type of access required.
The request by the application program to create shared buffer that already exists has an effect of checking the memory manager's specification for this buffer against the applications' specification and if the two agree the memory manager authorizes the application to perform subsequent read/write operations.

/* remove a circular buffer */
Boolean request_remove_buffer (String name, char *taskname)
Application program specifies parameters for the name of the buffer and the name of the requesting task. The request results in the name of the application being removed from the memory manager's list of tasks that are permitted to access the buffer. If the requesting task is the last on the list, the shared buffer is removed.

/* read from buffer */
Boolean request_read_buffer (String name, Address tolocaladdress, long records, int* recordsread, void workproc());
Application program specifies parameters for the buffer's name, the address where the data go to and the maximum number of records to be read from the buffer. The actual number of records read by the system is returned in **recordsread**. The application program can specify working procedure to be invoked regularly while the network message exchange continues.

/* write into buffer */
Boolean request_write_buffer (String name, Address fromlocaladdress, long records, void workproc());
Application program specifies parameters for the

buffer's name, the address where the data come from and the number of records to be written in the buffer. The application program can specify working procedure to be invoked regularly while the network message exchange continues.

The API functions return **Boolean** value **True** for successfully completed request and **False** otherwise.

The communication between the API procedures that are linked to the application programs and the memory manager has been implemented using standard TCP/IP connected sockets. The sockets operate in either blocking or non-blocking mode depending on the requirements of the application. An appropriate option is selected at the initialization stage of the DIME software ("b" option in command line parameters). The underlying communication between the distributed computing nodes is fully transparent to the application programs which see only a virtual shared memory as created through the API requests.

6. Typical traffic simulation and monitoring environment using DIME.

An example of a typical urban traffic simulation, monitoring and control environment is given in Figure 3.

The initiative for supplying the real-time messages and control data belongs to SCOOT-client application. All information is stored in a shared memory structure "buffer". This data can be retrieved by any client that has registered with DIME its intentions to read from the buffer. The simulation clients may also generate time critical data if, for

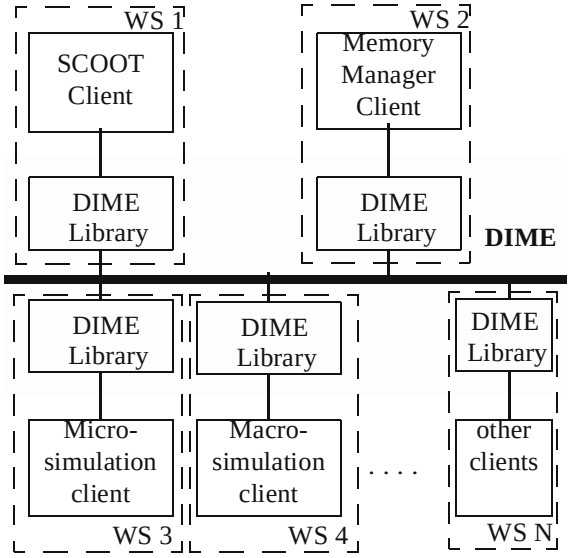

Figure 3. Using DIME in urban traffic simulation and control

example, they are interacting with another copy of a SCOOT software for the purpose of generating predictions of traffic evolution. In this case two buffers may be created: one, storing the simulated measurements, would be specified with a write permission to the simulation clients and a read permission to the SCOOT client, and the other buffer, storing the control responses to simulated measurements, would be specified with a write permission to the SCOOT client and a read permission to simulation clients.

The exchange of non time-critical information between the clients is performed via shared memory structure "array". In this case setting of read/write permissions facilitates the control over the information flow.

7. Simulation results.

DIME software has been evaluated in the context of urban traffic monitoring and control applications. The software was tested using both LAN and WAN distributed system configurations. The WAN configuration consisted of 4 nodes: one HP/Unix workstation at the Helsinki University of Technology and one PC DOS computer and two SUN UNIX workstations at the Nottingham Trent University. The software suite included two application tasks: an emulator of the traffic monitoring and control software (SCOOT) (this program essentially replayed historical SCOOT measurements and control data) and a macrosimulation module. The data used in the tests were collected by real-time control system SCOOT for the Mansfield traffic network (a town north of Nottingham). The shared memory manager task was placed on each of the four available computer nodes in turn and the application programs were monitored whether their requests for access to the shared data are being satisfied within the time-frame defined by the monitoring frequency. In all tests the data storage/retrieval rate by the application programs was better than 0.5 KB per second (requested performance by the SCOOT system for Mansfield region) including the case when both application programs accessed the memory manager task using WAN.

Additional experiments have been carried out to evaluate the maximum throughput of information for the DIME system. For these tests 4 workstations and a PC located at NTU, Nottingham and one workstation located at HUT, Helsinki have been used. The "worst case" scenario has been constructed by placing the memory manager task on a remote computer, situated at HUT and the application tasks on nodes at NTU. Consequently, every access to shared memory required network transmission. The "writer" task wrote data into the shared memory with a progressively increasing rate and the "reader" task attempted to keep-up with the "writer" retrieving all the data. The experiment was repeated with different number of "readers": from 1 reader in the simplest case up to 4 readers. The experiments showed that in

WAN working environment the maximum information retrieval rate was in the range 2.8-6.5 KB/sec. This performance did not depend substantially on the number of “readers” but it did depend on the network load. The maximum retrieval rate of 6.5 KB/sec was measured with 1 to 4 "readers" on a lightly loaded network and similarly the maximum rate of 2.8 KB/sec was measured with 1 to 4 readers with a busy network.

Another group of tests were carried out in a LAN working environment. Using the same suite of application tasks the performance of DIME has been found to be in the range 8-10.6 KB/sec. Over 99% of the time was spent on negotiating message delivery by TCP/IP and the remaining 1% on the actual shared memory accesses and housekeeping.

Figures 4a and 4b depict the respective performance envelopes for the LAN and WAN tests of the system.

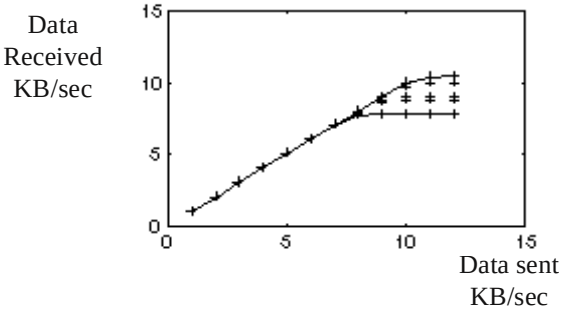

Figure 4a. DIME LAN performance for 2-5 user applications.

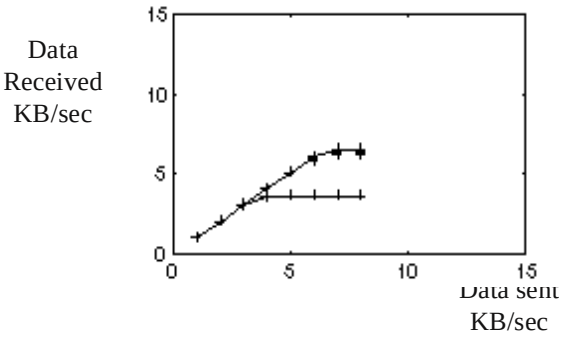

Figure 4b. DIME WAN performance for 2-5 user applications.

The tests confirm that the WAN configuration offers an adequate distributed traffic systems software development environment. However, because of the wide fluctuations of the data traffic over WANs, for the real-time operation of the system the DIME software should execute using LAN configuration, as originally intended. It must be pointed out however that this is not a limitation on the part of DIME and the use of more efficient WAN networks may well alter the conclusions.

8. Conclusions.

A distributed computers shared memory system has been successfully designed, implemented and tested in multi-client distributed environment. The data used in testing the system were provided by the real-time traffic control system SCOOT for the Mansfield region traffic network. The results show that the system could provide a generic processing harness for the execution of software modules of urban traffic control systems. The system is based on TCP/IP communication libraries and the design of the system reflects features specific for the traffic control. It is designed to support applications for both DOS and UNIX platforms.

Acknowledgments:

The authors would like to acknowledge the financial support of the EPSRC (GR/K16593) for Dr A. Argile and Mr E. Peytchev.

References:

[1] 3COM TCP/IP with Demand Protocol, Architecture Programmer’s Reference manual, Part No 8216-00, December 1990.

[2] Argile A.D.S., Distributed Processing in Decision Support Systems, Ph.D.Thesis, Nottingham Trent University 1995.

[3] Kosonen, I.; Pursula, M.; “A simulation tool for traffic signal control planning”. Third International Conference on Road Traffic Control. IEE Conference Publication Number 320. London 1990.

[4] L.Lamport “How to make a multiprocessor computer that correctly executes multiprocessor programs”, IEEE Trans. Computers, Vol C-28, No 9, Sept. 1979, pp 690-691.

[5] Peytchev E.T., Bargiela A., Parallel Simulation of City Traffic Flows using “PADSIM” (Probabilistic ADaptive SImulation Model), European Simulation Multiconference ESM'95, Prague, 1995

[6] Peytchev E., Bargiela A., Gessing R., “A Predictive Macroscopic City Traffic Flows Simulation Model”, European Simulation Multiconference ESM'96, Genua, 1996.