XDSM - an X11 based Virtual Distributed Shared Memory System.

A.D.S.Argile, A. Bargiela.

Department of Computing,
The Nottingham Trent University,
Burton St., Nottingham NG1 4BU, U.K,
arg@uk.ac.ntu.doc, andre@uk.ac.ntu.doc

Summary
An X11 based page-able shared memory system, permitting the implementation of a distributed water network monitoring and control software suite, is described in this paper. The system is based on the use of modular library and language specific interfaces, to access the underlying X11 communications platform supporting distributed shared memory (DSM). The system uses a server client relationship, with local computer node task managers, and uses the DSM for communication, configuration and coordination. An X11 based distributed mutual exclusion algorithm, based on the unconventional use of Lamport's bakery algorithm, is illustrated.

Keywords
X11, Distributed Shared Memory, XDSM, Bulk-synchronous parallel processing.

1. Introduction

When parallel tasks execute on a single processing node their data communication requirements can be provided by means of shared memory. This is because the shared memory paradigm gives the programmer a shared address space linking separate processes. It provides a logical view of data which abstracts out the coding requirements from the actual complexities of the physical data transfer.

Conversely, the conventional method of intertask communication via message passing, forces the programmer to be acutely aware of message source, destination, transmission protocol(s), and format. Thus message passing based communication can become quite complex in dynamically evolving software systems. This is especially true if there is no software layer translating the logical communication requests into physical hardware specific commands [9]. The rationale for shared memory has been stated explicitly by [12]:

> *"The shared memory system hides the remote communication mechanism from the processes and allows complex structures to be passed by reference, simplifying distributed application programming. Moreover, data in a distributed shared memory can persist beyond the lifetime of any single transient process."*

> *"The message passing models force programmers to be conscious of data movement between processes at all times, since processes must explicitly use communication primitives and channels or ports. Also since data in the data-passing model is passed between multiple address spaces, it is difficult to pass complex data structures."*

However, while the shared memory model of intertask communication simplifies program design, it

is limited by the power of the (single) host CPU. Increasing the available processing power means either using more than one CPU, or upgrading the host processor.

Using more than one CPU can give greater computation throughput by the exploitation of parallelism. However, closely coupled multi-CPU systems suffer from problems of scalability related to the complexity of the communication hardware required by CPU to CPU, or other hard wired connection topologies. Thus more general communication strategies based on networking software are desirable. The complexity of such distributed computing systems stems from the potential heterogeneity of CPU's, and heterogeneity of network communication protocols.

Distributed shared memory (DSM) is a concept of shared memory applied to loosely coupled systems, where true shared memory cannot be supported [10]. It attempts to combine the programming advantages of using shared memory for intertask operations, with the advantages offered by having tasks able to run on specialized hosts. Put more pragmatically [14]:

> *"Heterogeneous distributed shared memory (HDSM) is useful for distributed and parallel applications to exploit resources available on multiple types of hosts at the same time"*.

However, inherent problems associated with loosely coupled systems - architectural and communication heterogeneity, and complex considerations of data consistency [5][6][7], have been responsible for the lack of widely accepted implementations of distributed shared memory systems.

This paper presents a candidate implementation of distributed shared memory (DSM), which is designed for a network of heterogeneous computing nodes. The system described here was developed having in mind a particular class of industrial process control applications requiring extensive computational power, but involving only moderate interprocess communication. In particular, it has been used to implement a large software suite for real time water network monitoring and control, as explained in section 5.

The proposed DSM system is based on the use of the X11 Windows graphics standard. This offers the advantage of both portability and an integrated graphics environment for the development of graphics user interfaces.

2. X11 basics

X11 Windows is a network transparent, vendor independent graphics oriented operating environment [8]. It consists of two parts: an I/O server which controls the graphics hardware (display screen, keyboard and graphics pointing devices), and client application programs which require access to the visual display and pointer device, and/or keyboard. Clients use the display and other devices by sending message requests to the server and, when necessary, they receive replies back from it. The server also provides system and display information to clients, thus acting as a database.

This client/server relationship, which is based on an underlying message passing system, corresponds to the central server algorithm [12]. This is where global data is held by a central database task. This task, the server, controls client access to the global data by supplying data on request to clients, and it updates the global data when sent new data by clients. Thus global data and client data may be structured in totally different ways. Furthermore, in the algorithm, global data does not migrate to other servers.

Figure 1 illustrates the concept of the X11 client server arrangement. An X11 server provides high level access to display and keyboard, and globally available client and system data. Clients access the I/O devices and data by sending requests to the server via a suitable network communication protocol. Clients that are local to the X11 server normally use a local communications protocol.

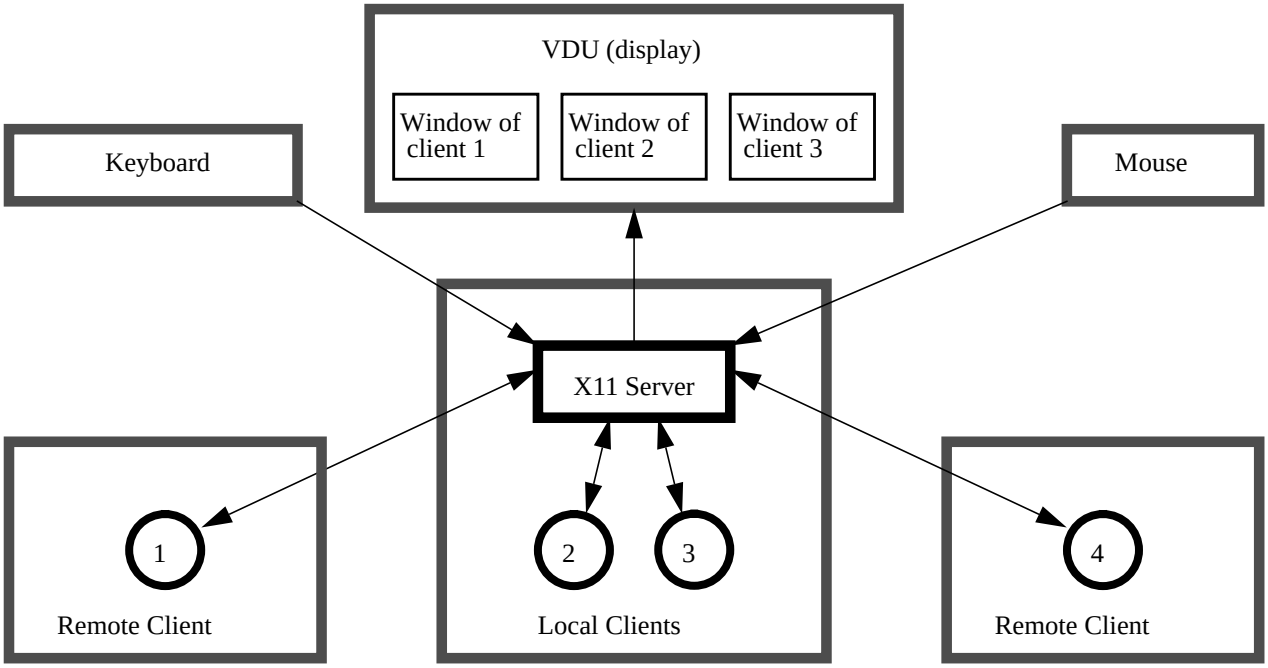

Figure 1. X11 client server arrangement

Access to the display and keyboard, together with the organization of the display, is provided by the windows structure. Clients are notified of any change to the windows by the server generating appropriate events. It is the responsibility of the client to monitor the event queue and to act accordingly. The communication of data between clients and a server is effected by means of properties, which are data structures associated with corresponding windows. The following gives a summary of the key concepts underlying the X11 Windows system:

- Windows

Windows are normally rectangular regions of the screen serving as output for client graphics and input from the keyboard. Windows are selected using a graphics pointer device, and they provide applications (clients) with a method of organizing the display screen. Each display screen consists of a single root window covering the entire screen. This serves as parent for overlying child windows. These child windows may also have child windows, and be overlying or underlying other windows. When a client connects (registers) with a server, it is assigned its own ’private’ window within the server’s display. The client may then create more child windows. This relationship between parent and child window creates a hierarchy of windows.

- Events

X11 clients frequently need to be aware of what is happening to any windows created for them, or about other windows they may have an interest in. Clients have two ways of obtaining information about the window environment. For immediate needs they can explicitly request specific window information from the server. For continual change notification they instruct the server to asynchronously send status change information. This status information is contained

in event data structures, which are queued on an input event queue.

Various window event structures exist to provide status information about window size and stacking changes, window property changes, pointer position and motion, client messages, etc. In addition, timer and I/O event facilities are defined by the standard X11 toolkit. X11 clients are in effect event driven, because they are expected to continually monitor incoming window and I/O events, and then react to them.

- Properties

Properties are data structures maintained by the X11 server and referenced by display and window id. If the window is destroyed the data will be destroyed by the server. The properties are structured as 1, 2, or 4 byte arrays permitting the storage of text strings and integers (which can be mapped to ASCII floating point values). Properties are referenced by host display, window, and property id. Thus a property stored in any window may be accessed by any client connected to the window's server. So properties represent globally sharable data.

Properties can be owned exclusively by a client task; this can provide the basis for an access locking protocol. A client wishing to gain privileged access to the property determines if the property is unowned, and then sets the ownership of the property to the window of its choice.

The client-server communication in X11 is based on a message passing system which is accessed via a Remote Procedure Call (RPC) interface. This means that the underlying detail of physical message routing is hidden from the user [4][10]. The mechanism used to communicate graphics function requests between client applications and the graphics server can be adapted to communicate appropriately structured, shareable properties. It is this mechanism which is utilized to support the facility of the distributed shared memory.

3. X11 based Distributed Shared Memory (XDSM)

A fundamental development for the distributed shared memory system is the definition of shared data areas and their mapping onto the physical memory resources. In our system, the local memory of the client is treated as addressed data blocks to be stored in global, shared X11 properties. This principal characteristic of our system is reflected in its name: the X11-based Distributed Shared Memory system - XDSM.

XDSM is constructed on the basis of library modules. Various modules exist for creating GUI interfaces, user language code interfacing, and GUI message generation. Figure 2 illustrates the logical information flows between the X11 server and main XDSM modules. At the center of an X11 client there is an endless loop checking the input event queue for incoming information. Events are extracted and processed by XDSM library code on the basis of whether they may be intended for the X11 GUI toolkit, or are control area property change events required by the control module. Reception of relevant property change events causes the configuration and control module to instruct the data access module to copy XDSM control data to a local copy of the control area.

- Shared data access module.

The most important module for the XDSM user is the module dealing with shared data access. This operates the various XDSM access protocols needed when reading and writing shared data, and transfers data directly between local memory, accessible to application code, and the XDSM maintained by the X11 server.

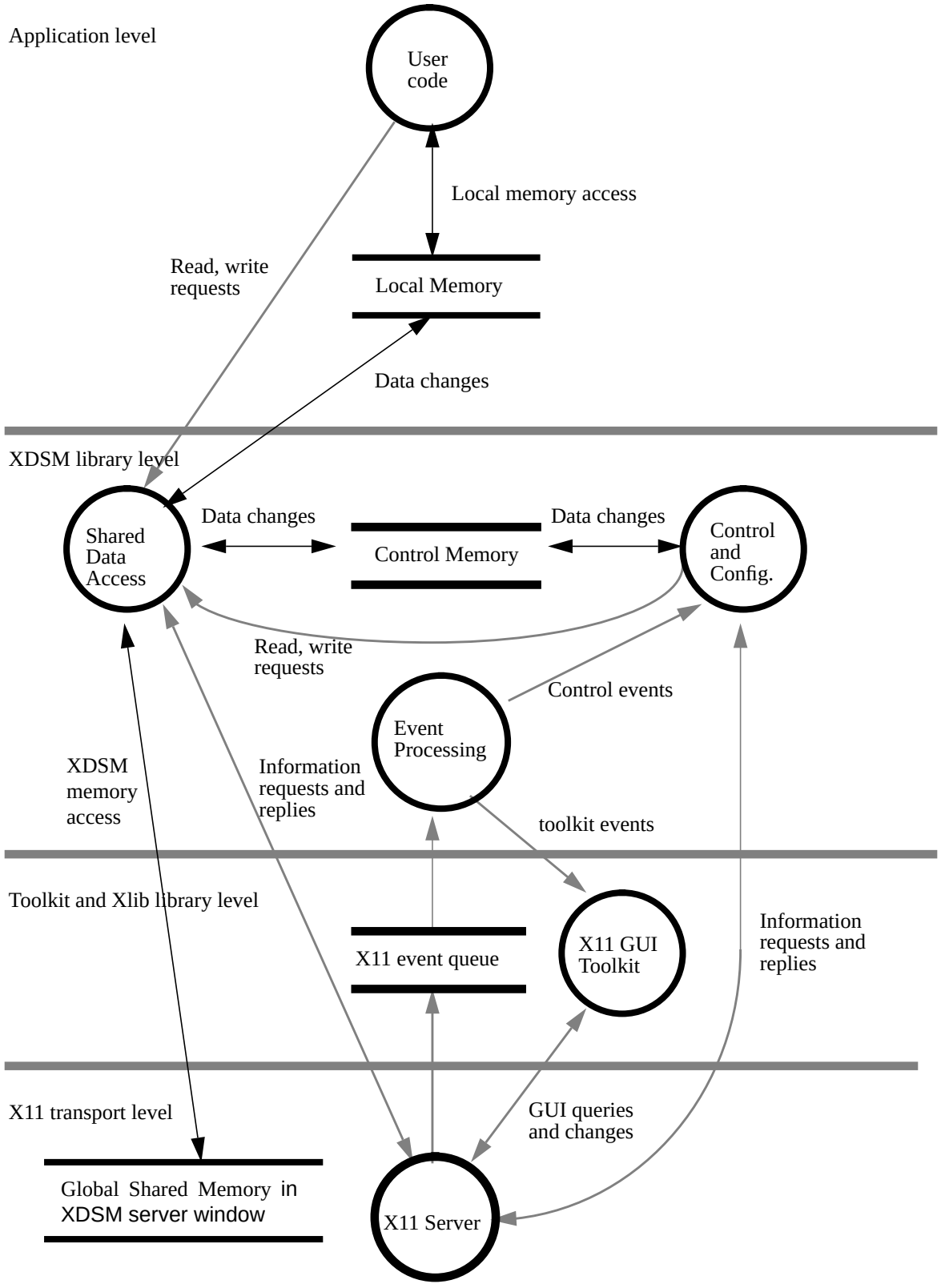

Figure 2. X11 and XDSM event processing

By using a property access locking protocol (explained on in section 4), X11 is used to supply all required memory access operations - lock, free, read shared area(s) to local memory(s), and save local area(s) to shared area(s) - in both access blocking and non-blocking forms.

- Event processing module.

The event processing module reads the queue of events sent by the server, and directs appropriate events to the X11 toolkit and to the XDSM control module. The X11 server, user code and the XDSM library tasks, execute in a pseudo-parallel manner. The event processing module implements in effect the scheduling of these tasks.

While the event processing loop is an integral part of any X11 application, for the purpose of the distributed shared memory system it is seen as an implementation detail. For this reason, the event processing module is embedded within the XDSM system and is transparent to the distributed applications.

- Control and configuration module.

The control and configuration module is responsible for maintaining the integrity of the distributed processing system. The module maintains a control data area which contains task pathnames, status and command information. The maintenance of the control data in XDSM systems is linked to X11 event processing and is hidden from the application programmer.

To prevent unnecessary re-reading of the control data, the XDSM system uses the facility by which X11 is instructed to inform an application of any modification to window properties. Changes to the control area are reported in the input event queue, and extracted from it by the control module. This passive approach to monitoring data consistency avoids the round trip delays incurred when requesting information from the X11 server - the message passing involved in a client→server→client information request requires a minimum delay of 2 task re-schedulings.

Regarding notification of property changes, there is evidence that occasionally events do not arrive, or can be seriously delayed. Therefore, using this passive communication protocol to support more complex XDSM page operations, corresponding to read, write, and delete faults, and dynamic data merging to give continual local/remote XDSM data consistency, is problematic. Implementing continual consistency checking and updating by data merging (see section 4 on Paging) would involve direct event processing by the data access module.

4. XDSM Implementation Issues

- XDSM configuration and start-up.

To provide for XDSM start-up and global control, an XDSM supervisor task was created. This provides a convenient window for XDSM data, and can be used to monitor the ownership status of each data area, periodically save the XDSM to disk, and provide various start-up and control options. However, its most important function is to start-up the XDSM system. To do this it reads a local configuration file which provides:

- A unique XDSM identifying name for the task suite to use.
- XDSM host names, and the operating system in use.
- The names of the local hosts together with the names of the local host task managers, and

which tasks are to run.

The supervisor (XDSM server) remotely executes each local host manager (this is operating system dependent, and may have to be done manually). The XDSM display and identifying name are supplied as command line parameters. After locating the XDSM control area and obtaining the client pathnames from it, the clients are activated on their associated hosts. The clients then put their status information in the control area. Should a client fail, the loss of its window can be detected by the manager so that it can be re-executed.

• Control area facilities.
To deal with task control and XDSM configuration, an XDSM control module and memory area has been provided (shown in figure 2). This contains lists of hosts, pathnames, status and command data for managers and clients. The command information relates to general commands issued at user instigation by the supervisor - start, stop, exit. The status information relates to a client's executing task status. By default, clients are marked as absent. When the client task is initiated it is marked as loading, rather than running. This is because task execution may fail, but the initiating manger will not be informed by the operating system. Also, there is an undeterminable delay before the client can locate and update the XDSM control area. Therefore, between the absent and the executing state of a task, there exists an intermediate state of loading. Once a client has connected with the XDSM, it marks itself as either running or waiting, as appropriate.

• Mutual exclusion during XDSM data access.
When requiring sole access to shared data, a client must determine if the property is unowned. When it is, it must set ownership of the property to its own window in the display where the required XDSM data is located. However, because X11 is asynchronous, with variable, buffered message delays, it is possible for applications in contention over data ownership to incorrectly gain multiple ownership.

An attempt to provide mutual exclusion by implementing a server grabbing protocol, which involves a client gaining exclusive access to the X11 server, is also deemed to be unsatisfactory as it can lead to the starvation of individual clients. The current version of XDSM implements a separate mutual exclusion protocol which is based on the use of a standard non-distributed mutual-exclusion algorithm. The property based protocol is a variant of Lamport's Bakery Algorithm [2] [11]. It uses property data to provide an abstraction of flat memory data structures (queue position numbers). The protocol causes the client at the head of the queue to wait in the queue until the data is free. It is implemented by the following stages:

1. Assign queuing number (highest existing number+1) and enter the queue.
2. Wait until there are no smaller numbers in the queue.
3. Wait until data is free.
4. Lock data.
5. Delete queuing number - exit queue.

The queuing system inherent in the Lamport Algorithm resolves this potential liveness problem for the client tasks. This is because after a process enters the queue, it will eventually move to the head of the queue, whereupon it can gain exclusive access to shared memory data.

• XDSM consistency and access efficiency.

Properties are also used to provide memory consistency checking. Each data property is associated with the ’last accessed’ property. Each process that accesses the data sets the property ownership to its window. Whenever the area is to be read, this is checked to see if the data in it has been modified by another application. By making it unnecessary to re-read unchanged data, the XDSM system increases the efficiency of the shared data access.

• Data paging.

Because of restrictions in the way X11 properties can be used, modifying one item of shared data requires that all the data be rewritten. This implies that if the size of the properties are big, there is a large communication overhead associated with the update. XDSM overcomes the problem by automatically segmenting the data into smaller ’pages’. Storing data as pages has the advantage of reducing the amount of data to be read in and out, and by using the ’last accessed’ property mentioned above, unnecessary reading can be avoided entirely. If the task holds a shadow copy of the data originally read from the XDSM, this can be compared with local memory data, and only modified pages saved back to shared memory. Merging data changes made both in local memory and XDSM, followed by updating the XDSM, is also possible.

5. Application

Using the XDSM, the following functionality can be provided for an application suite:

1. Definition of shared data areas - with user monitoring and control of the shared data provided by an interactive GUI graphics user interface.
2. Shared or exclusive access to global data.
3. Task coordination and control, including automatic task start-up and local task control.
4. Distributed error handling and recovery - achieved by maintaining shadow copies of the shared data, and by error handlers.

XDSM has been used to facilitate the distributed implementation of a realistic decision support system for the monitoring and control of water distribution networks. The structure of the application software suite is shown in figure 3. The suite comprises network and telemetry simulators, state estimators and an operator interface, configured to provide a classical feedback control loop. Other modules, which are scheduled for future incorporation into the suite, are concerned with optimal control and telemetry confidence limit analysis.

The original application suite was implemented as a set of concurrent tasks communicating by conventional shared memory. The parallel program implementation of this system is a natural extension of the original single processor multitasking implementation [3] [1]. XDSM's main addition to this system, though not involving any change to the original subsystem concept, is the XDSM supervisor task, together with local host task managers.

Using a network of 4 Sun SC Sparcstations running the simulation, telemetry, estimation, and operator interface modules respectively, the cycle time achieved for a 65 node network was approximately 10 secs. This is at least an order of magnitude better than would be expected in real life. Projecting the results for larger networks, it is expected that the communications overhead will, at worst, increase linearly with network size, so the communication to computation time ratio will actually decrease.

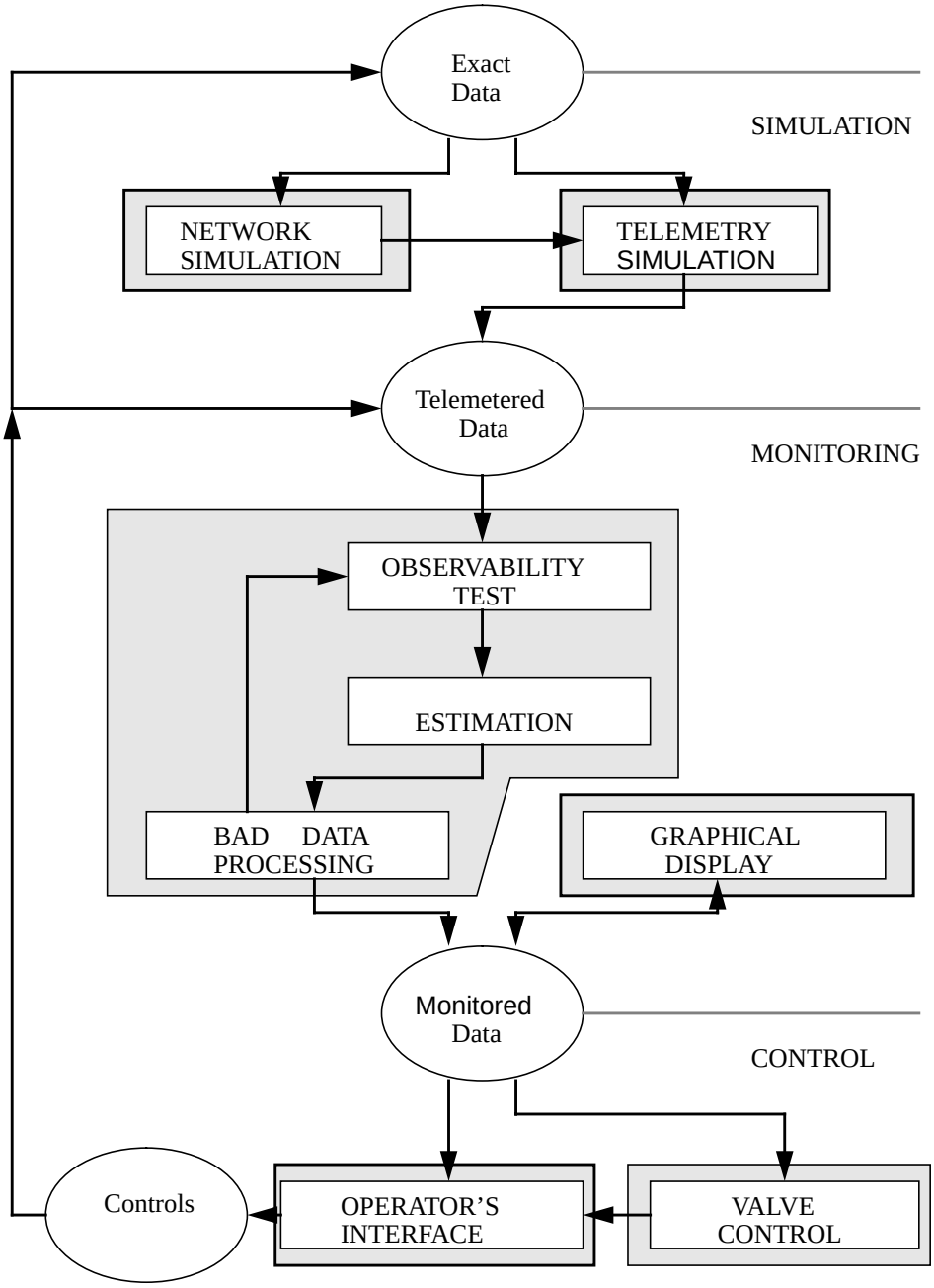

Figure 3. Water network monitoring suite

6. Conclusions

Our work to date has resulted in the development of an original software product, an X11 based Distributed Shared Memory (XDSM) system, intended for applications in scalable heterogeneous distributed computing environments.

The XDSM system has been successfully used to provide an implementation framework for a telemetry system concerned with the monitoring and control of water distribution networks. The performance of the current implementation of the XDSM indicates its applicability to such a class of industrial process control applications.

As major performance differences relating to various mutual exclusion algorithms have been noted, the next stage of our research will be concerned with performance evaluation. This will be done in conjunction with the development of an orthogonal programming meta-language harness in-order to support the implementation of the Bulk Synchronous Parallel (BSP) model of parallel processing [13].

7. References

[1] Argile.A. & Bargiela.A., Using X11 windows to provide shared task-memory in distributed systems, in Integrated Computer Applications in Water Supply, Volume 1, Coulbeck.B.(ed.), Research Studies Press, 1993.

[2] Ben-Ari.M., Principles of Concurrent and Distributed Programming, Prentice Hall, 1990.

[3] Bargiela.A. & Al-Dabass.D., A Simulated real-time environment for verification of advanced water network control algorithms, Systems Science 14.3. 1988.

[4] Coulouris.G.F. & Dollimore.J., Distributed Systems, Concepts and Design, Addison-Wesley, 1988.

[5] Heddaya.A., & Sinha.H,
An implementation of MERMERA: A Shared Memory System that Mixes Coherence with Non-Coherence, BU-CS-92-013, 1992.

[6] Heddaya.A., & Sinha.H,
An overview of MERMERA: A System and Formalism for Non-coherent Distributed Parallel memory, BU-CS-92-009, 1993. To be published in Proc. 26th Hawaii Int. Conf. Sys. Sci.

[7] Hutto.P.W. & Ahmand.M., Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories, IEEE 10th Int. Conf. Dist. Systems, 1990, 302-307.

[8] Jones.O., Introduction To The X Window System, Prentice Hall, 1989.

[9] Kranz.D., Johnson.K., Agarwal.A., Kubiatowicz.J., Beng-Hong.L., Integrating Message Passing and Shared Memory: Early Experience, SIGPLAN Notices, 1993 28(7), 54-63.

[10] Levelt.W.G, Kaashoek.F., Bal.H.E., Tanenbaum.A.S., A Comparison of Two Paradigms for Distributed Shared Memory, Software-Practice and Experience, 1992, 22(11), 985-1010.

[11] Raynal.M., Algorithms for Mutual Exclusion, North Oxford Academic, 1986.

[12] Stumm.M. & Zhou.S., Algorithms Implementing Distributed Shared Memory, COMPUTER, 23, 5, 1990.

[13] Valiant.L.G., A Bridging Model for Parallel Computation, Comm.ACM, 1990,33.8, 103-111.

[14] Zhou.S, Stumm.M. & Wortman.D., Heterogeneous Shared Memory, IEEE Trans. on Parallel and Distributed Systems, Vol. 3, 5, 1992.