Using X11 Windows To Provide Shared Task Memory in Distributed Computer Systems

A.D.S.Argile & A.Bargiela

1. Abstract

An X11 based shared memory system, permitting the implementation of a distributed water network monitoring and control software suite, is described in this paper. The application source code is in FORTRAN and accesses shared memory via a C library module requiring no other support packages. X11 based distributed mutual exclusion algorithms, based on the unconventional use of Lamport's bakery algorithm, are evaluated. The system is compared with other methods of providing distributed shared memory.

2. Introduction

The distributed shared memory system described here was developed to facilitate the transfer of software for real time water network monitoring and control, from a single CPU to a distributed computing environment. Because of inherent parallelism, the original application was implemented as a set of concurrent tasks requiring shared memory resources [2].

When such tasks reside on a common CPU, shared memory can be provided by either memory control logic or software. This permits the solution of parallelism either by splitting the load between parallel, concurrent, modules, held within a single task, or by concurrent tasks running under a concurrent operating system. But this approach, which simplifies both program maintenance and the operation of complex algorithms, is limited by the power of a computer's CPU. Increasing the processing power available means using more than one CPU, ideally one per task. This also has the advantage of permitting individual tasks to execute on more suitable host architectures (giving mathematical tasks access to high performance floating point processors, graphics tasks access to graphics workstations, etc), or on hosts more suited to the user's needs.

The X11 graphics and networking package can be used to provide distributed shared memory [3], together with 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. Unlike operating systems targeted at single processors, an X11 based shared memory system permits shared memory to be accessed over a distributed network. An X11 based access protocol prevents simultaneous access (via mutual exclusion) of the data area(s).

X11 is a flexible operating environment - by using the terminal windows generated by the xterm application tool, a single terminal can run and simultaneously display many interactive applications, either together or with their I/O routed to alternative terminals. Thus an entire application suite can be initially developed and tested on a single CPU. An X11 based shared memory system can be tailored to systems where the key system components - host machine(s) and network links, are liable to fail without warning. Furthermore, X11 is widely available on a variety of hardware platforms.

3. Basic Methodology

The principle of shared memory depends on the use of X11 properties. These are byte or integer data arrays linked to the X11 window of the task that created the data, and can be viewed using appropriate X11 software tools. Properties are identified by means of X11 atoms (4 byte unsigned integers). Atoms are assigned unique values by the local host X11 request coordination and device driver (the server). When an application (the client) requests that the server creates an Atom, it provides the server with a unique identifying text label for the Atom. Once created, atoms cannot be deleted or renamed.

A property may be owned by a window, which is normally a viewable graphics area maintained by the X11 server. Windows are logically linked together by the server into a window tree; there is normally only one window tree per host. Property ownership means that all data identified by the property atom is now 'owned' by the window, even though the data does not 'move' from its original window. However, ownership does NOT prevent (lock out) any other task from modifying the data. Even so, property ownership can provide the basis for the access locking needed to permit data to be used by only one user at a time (access synchronization). However, it must be noted that although property data can be accessed over a distributed network, it cannot be owned over the network. This is because properties can only be owned by windows that are in the same window tree as the creator window. The data blocks (areas) were organized as 32 bit long

integer arrays, the size of which is practically unlimited (disk memory limit) [4, 5].

A provider task initializes data areas in one (or more) windows. This/ these windows have specific identifying names so that another task (the user), even on a different host, can locate the data window(s). In the simplest form of shared memory, this is all that is required. In our implementation, the provider task also facilitates saving and restoring data from disk, which can involve locking the areas against user access. Figure 1 shows how a provider task and a user task would interact to share data, and how they interact with a user.

A protocol was developed which uses the fact that the X server has the capability to grant ownership of properties to windows. As indicated above, this is not a true lock and key situation, because the server does not require the owner to free a property before granting ownership to another window. Thus the protocol provided must be strictly adhered to. Specific routines were developed to establish X11 display connections to host displays and for searching X11 window trees for the data window(s).

Figure 2 illustrates the process by which data area ownership is granted to a task. It was found that the asynchronous nature of the X11 request processing system, where replies from the X server can be out of sequence, resulted in occasional errors when locking distributed data areas (see Figure 3). In our implementation this problem has been overcome by the use of mutual exclusion algorithms. The nature of X11 permitted the use of simpler non-distributed algorithms [6, 7], rather than the use of relatively less robust, message passing, distributed mutual exclusion algorithms [6].

4. Mutual Exclusion

4.1. Why Mutual Exclusion was Required

In the initial implementation, areas were locked by assigning ownership of their properties to the requester. In order to ensure that ownership had been granted, the server was interrogated for the identity of the new owner (see Figure 2). This is standard X11 programming practice for checking if ownership has been successfully regranted [11, 12]. It was found that occasionally, when several processes were competing for ownership, two processes would be informed by the server that they now had ownership. This means that the standard ownership checking method is invalid.

Because X11 is normally asynchronous and clients are time sliced by the scheduler in an operating system dependent way, it is possible that when clients compete for an area, ownership is granted to one client, and is then regranted to a client which did not detect the prior ownership change due to

delays in server and network operation (see Figure 3). Providing client requests with explicit timestamps to ensure that delayed requests are ignored by the server [4, 5], does not prevent this type of error. To overcome the ownership problem, access to the instructions for changing property ownership was restricted with a mutual exclusion protocol.

4.2. Implementing Mutual Exclusion

Various mutual exclusion algorithms are described by Ben-Ari [6] and Raynal [7]. We required simple, thus maintainable and modifiable, mutual exclusion protocols. Therefore a true distributed mutual exclusion protocol [6, 8] was rejected because it did not allow for the possibility that processes will fail, and like the message passing based protocols, there are extensive intertask communication overheads. The other mutual exclusion protocols were not suitable, since they required indivisible operations and did not easily generalize for *n* processes [6, 7].

Mutual exclusion protocols either use a 'centralized entity' such as a memory location or clock, or state variables, or message passing [7]. Because X11 allows data to be communicated between tasks on different architectures, it allows distributed tasks to read and write non distributed data structures. These can be used to implement the state variables (tickets) for Lamport's Bakery Algorithm [6, 7]. The algorithm's underlying principle has been described as: “In the bakery algorithm, a process wishing to enter its critical section is required to take a numbered ticket whose value is greater than the values of all the outstanding tickets. Then it waits until its ticket has the lowest value. ...there need be no variable which is both read and written by more than one process” [6]. Several practical considerations were applied when selecting a mutual exclusion protocol derived from Lamport's Algorithm:

1. Efficiency - will use of the protocol slow the client down too much?
The Lamport algorithms use what is effectively a FIFO queuing system. This ensures that access to the data areas is fair [6]. Strong or weak fairness then depends on the chosen lock implementation. Peterson's version of Lamport's algorithm [6, 7, 10] uses bounded ticket values, but was found to be slow (due to it's complexity and the large number of X11 calls for ticket data). Of the algorithm's needing unbounded ticket values, Lamport’s [6, 7] original bakery protocol was about 50% faster than Peterson's algorithm, while Hehner and Shyamasundar's algorithm [7, 9] was simpler to implement, and slightly faster (the speed difference was small).

2. Reliability - can it cope with the dynamic loss and gain of clients?

Most of the mutual exclusion algorithms assume, or require, that all clients are functioning during execution of the software suite. However, in our water network system application suite, the number of executing clients can vary with time. In our implementation of Lamport's algorithm, because new processes attain ticket numbers by checking all existing tickets, existing processes are not affected by new processes joining the access queue. Thus processes can start up at random. Also, the dynamic loss of clients is permitted, on the basis of checking if a ticket property is owned by a client. Tickets with no owning window can be ignored.

3. Codability & Maintainability - these are direct opposites.
The more involved the protocol, the more involved the coding, so the less maintainable (and reliable) the code. Peterson's algorithm, as intimated above, was significantly more difficult to implement, while Hehner and Shyamasundar's algorithm was simplest to implement.

5. Similarities with other Shared Memory Systems

Research into distributed shared memory extends back more than ten years. The conceptual techniques employed have been overviewed by Bal & Tanenbaum [13]. Our implementation has some similarities to:

1. Problem-oriented shared memory [15]. Data memory is organized to be application specific, and is accessed by commands to fetch, store, lock and unlock data. The semantics of the fetch and store commands have similarities to the X11 get and change property commands. Data is maintained by memory control modules located at specific network nodes. To improve efficiency, data may be cached locally. Because a read is not always preceded by the most recent write this creates a coherency problem.

2. Agora shared memory [16]. This uses shared data structures accessed by library routines. Data structures consist of immutable typed data (records) accessed via mutable maps. Data is modified by the addition of new data, with corresponding map amendment; the defunct data is eventually reclaimed (garbage collected) [13, 16]. The (Mach) operating system makes the single copy of data held at a given node accessible to all tasks within the node's address space. Other nodes receive a copy of the data. Thus there is a difference in data access between tightly and loosely coupled architectures. The modification of data held at a node is propagated by specific Agora tasks to all other nodes using a copy of the data. Though there is access locking of the master copy, memory may not be coherent [16]. Data format conversion - byte swapping (as in X11), bit shuffling, etc

- for different processor architectures is supported. This is not the case in problem-oriented or virtual shared memory [13].

3. Shared virtual memory [14]. Memory is organized as fixed size pages, and made accessible to all processes by local memory mapping managers using a memory coherence protocol. As this system is addressed like true virtual memory, data is not copied by explicit command, but implicitly upon memory reference. This requires the use of a hardware Memory Management Unit [13, 14, 17].

If implementation details are ignored, the closest similarity is with problem-oriented shared memory - application specific data is accessed via specific commands and may be stored at numerous locations, with X11 servers acting as the memory control modules. Agora has two partial similarities, automatic data format conversion (to a higher specification), and data access via library routines (only needed for inter-node shared memory). Shared virtual memory, while superficially similar, functions in a quite different manner, with direct (virtual) memory access to a single node data copy, as in Agora.

An operational problem with distributed memory is ensuring that data is consistent [13]. In shared virtual memory, consistency is guaranteed because on accessing a new memory page, existing copies of the old page are invalidated, so that data is never allowed to get out of date. Such a system can lead to 'thrashing' , and considerable demands on the system. Problem-oriented and Agora shared memory have no fixed method for dealing with lack of consistency between different copies of the data; they may require the applications programmer to provide a solution.

In our current implementation, so long as the routines using mutual exclusion are used, data can only be accessed by first locking the data. Locking the data automatically causes reading of the data. At some point the data must be freed, causing an automatic write. Thus local data copies only become stale once unlocked. This data will then be unwritable. So memory coherence is guaranteed providing that the access protocol is adhered to. However, this requires that the application programmer notes the implications of unlocked data being invalid data.

6. Industrial Experience

The distributed processing harness outlined above has been applied to the water network monitoring system described in detail in [1] and [2]. The overall structure of this software suite is given in Figure 4. The software essentially consists of three subsystems concerned with simulation,

monitoring and control of water networks. The subsystems are configured so as to provide a classical feedback control loop. Each subsystem is composed of self-contained modules performing specific tasks.

Considering first the subsystems: the monitoring tasks expect to receive telemetry data and are entirely decoupled from the software that actually provides these readings. In one case, as illustrated in Figure 4, the data might be provided by telemetry simulation software, which corresponds to 'operator training' or 'off-line assistance' scenario. But in 'on-line' operation the telemeasurements would be provided by telemetry hardware. This decoupling of subsystems enables different frequency of access to the telemetered data by the simulation and monitoring software, subject only to the mutual exclusion of data update operations.

Within the subsystems, the individual tasks are also largely autonomous. For example the observability test which is responsible for verifying whether the measurement set is sufficient for subsequent computations, may run with a frequency different to that of the state estimator which utilizes the verified measurement set. Indeed, the system allows for more than one state estimator to run concurrently (e.g. to achieve better error rejection capability), or alternatively, temporarily there may be no state estimator running.

The modular structure of the software, quite apart from its software engineering rigor, emphasizes the concurrent nature of processing tasks that occur within the context of water network monitoring and control systems. In the original implementation [2], the concurrent processing tasks were mapped onto pseudo-concurrent processing of a single CPU running under a multitasking operating system. Each task communicated with others through shared memory areas with specified access privileges and the timing of task execution and synchronization was achieved by reference to semaphores and event keys in shared data.

While the conceptual design of the system is not compromised by a single processor implementation, it is clear that the gradual increase of the complexity of individual tasks (historic trending of telemetered data, data management, optimization of control etc.) leads to the specialization of processing and the introduction of distributed processing nodes. Furthermore, the distributed processing concept promises ease of expansion into areas which are complementary to on-line operations such as medium and long term system planning, system management etc.

The distributed processing harness described in this paper offers a flexible framework for mapping individual tasks onto the most appropriate network of processing nodes and, most importantly, it provides a physically distributed logical shared memory. The underlying mechanisms for

creating and accessing such shared data areas are hidden from the user. Therefore the application program's view of such areas is, as in a single processor implementation [2], a 'common' block. The only demonstration of a distributed implementation of the shared memory is reduced efficiency of access to data, due to the need for explicit network transfers. However, in the case of a water network monitoring system, it has been found that the reduction of data access efficiency is not critical since the interprocess communication is relatively infrequent and involves small amounts of data while, at the same time, the distribution of processing benefits the computationally intensive modules.

Using a network of 4 Sun SLC 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.

The main addition to the system, though not involving any change to the original subsystem concept, is the provision of a data pool task. This holds the shared data areas in one of its windows. This task also monitors the ownership status of each area, and periodically saves the data areas to disk. It also has system control options to save the areas to disk, to lock all areas and reinitialize all shared data. The subsystem tasks all have a similar visual representation, presenting the user with a standard graphical interface.

7. Conclusion

The key aspects of the X11 shared memory model implemented here are:

1. All low level operations supporting shared memory are performed by X11, freeing the conceptual model from implementation considerations.

2. There is no difference in data pool access for tasks on tightly or loosely coupled architectures.

3. Shared memory is guaranteed to be coherent if an access locking protocol is used. This is not always so in cached systems [14], or asynchronous systems such as X11.

The main advantages of using X11 Window properties to provide distributed data are:

1. If the data is stored in a display's root window, the data will exist for as long as the X server is functioning. No program is needed to maintain

the data, although specific applications can be used to initialize, remove, and periodically dump the data to disk.

2. The protocol developed here will run on any system supporting X11 Version IV.

3. In distributed systems, failure of one node in the system can lead to the loss of critical information. Here the danger is reduced because of the use of a central pool, which can be periodically saved to disk.

4. If required, the loss and gain of clients can be detected by monitoring client specific properties.

Because of restrictions in the way X11 properties can be used, there is a relative loss of performance; modifying one item in a shared area requires that all data items be rewritten. A synchronization call is then required to ensure that the data has been updated before further changes are made; synchronization calls reduce performance [4, 5, 12]. Together with the use of a mutual exclusion protocol, which involves the overheads involved in inter task arbitration, it means that modifying data areas is a relatively slow process. Performance will be adversely affected by an increase in both the level of network traffic and demands on the server. It could also be affected by the efficiency of disk access, since on some systems such as UNIX, it is likely that the server will store some property data on the disk (virtual memory).

An X11 based system can be designed to give protection against loss of the data pool, and the dynamic loss or gain of clients. While it is not suited to the frequent transfer of 'large' quantities of data, it has proved an adequate vehicle for our water network monitoring and control system. This is representative of a class of applications requiring a reliable data pool, where individual tasks may be terminated at will, and where there are varied networked host architectures.

The basic development research into X11 based distributed memory is now complete. The next stage will be the implementation of a distributed task control system. Continuing development of the water network program suite requires that studies of communication efficiency be undertaken, with the possibility of a major revision of both the mutual exclusion protocol and memory access methodology at a later date. Linda’s 'tuple' memory space [18] has been used to support distributed shared memory [13] in a way which provides implicit mutual exclusion. It should be possible to extend our system to access the shared data pool as if it were 'tuple' space. This will be implemented to allow comparative efficiency studies.

8. References

[1] A.Bargiela, On-line monitoring of water distribution networks, Ph.D., Thesis, University of Durham, May 1984.

[2] A.Bargiela and D.Al-Dabass, A Simulated Real-Time Environment For Verification Of Advanced Water Network Control Algorithms, Systems Science, No. 3, Vol. 14, 1988.

[3] A.D.S.Argile, An Investigation of X11 based Intertask Communication and HCI in Water Network Simulations, M.Sc., Sept.1991, Nottingham Trent University.

[4] R.W.Scheiffer, J.Gettys, R.Newman, X Window System, C Library and Protocol Reference, Digital Press, 1988.

[5] Nye.A., The X Window System, Volume 2, Xlib Reference Manual, 1990 O'Reilly & Associates,Inc.

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

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

[8] Ricart & Agrawala, An Optimal Algorithm for Mutual Exclusion in Computer Networks, Communications of the ACM. 1981.24.1.

[9] Hehner & Shyamasundar, An implementation of P and V, Information Processing Letters, 1981.12.4.

[10] Peterson, A new Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables, ACM Transactions on Programming Languages & Systems, 1983.5.1.

[11] Barkaki,N., X Window System Programming, 1991 SAMS.

[12] Jones.0., Introduction To The X Window System, 1988 Prentice-Hall.

[13] Bal.E.H., & Tanenbaum.A.S., Distributed Programming with Shared Data, Comput.Lang. Vol.16, No.2, 1991.

[14] Li.K., IVY: A Shared Virtual Memory System for Parallel Computing, Proceedings of the 1988 International Conference on Parallel Processing, Pennsylvania State University Press,1988.

[15] Cheriton.D.R. Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems. Operating Systems Review, 1985.19.4.

[16] Bisiani.R. & Forin.A., Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems., Prod. 2nd. Int. Conf. on. Architectural Support Programming Languages & Operating Systems., 1987.

[17] Li.K. & Hudak.P., Memory Coherence in Shared Virtual Memory Systems, ACM Transactions on Computer Systems. 1989.7.4.

[18] Carriero.N., & Gelernter.D., How to Write Parallel Programs: A Guide to the Perplexed., ACM Computing Surveys, 1989.21.3.

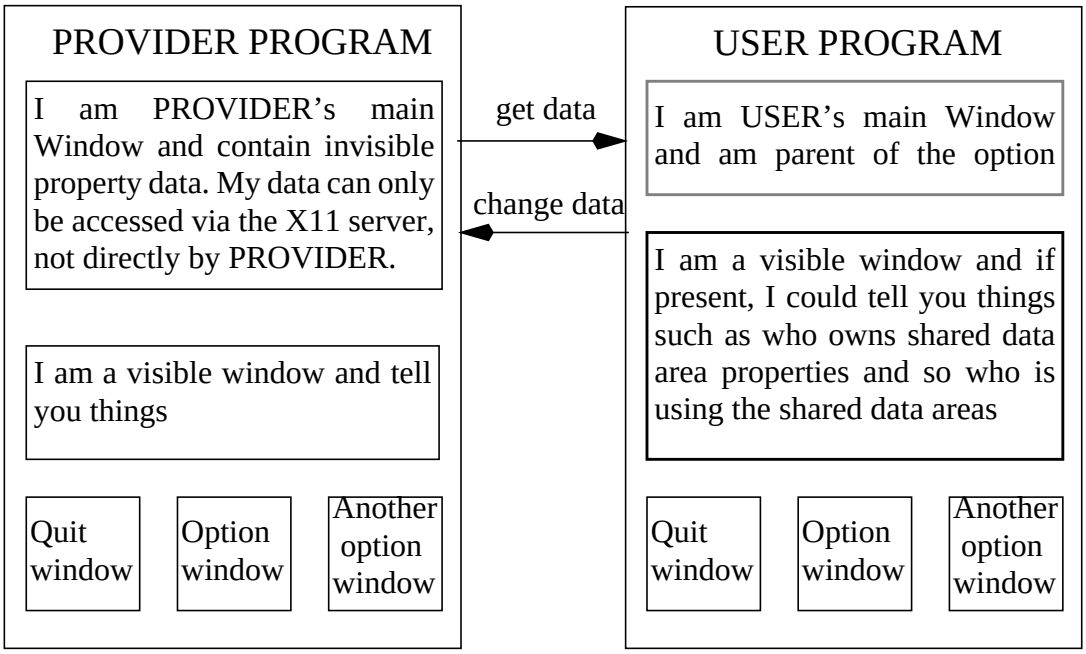

Figure 1. The Provider and User relationship

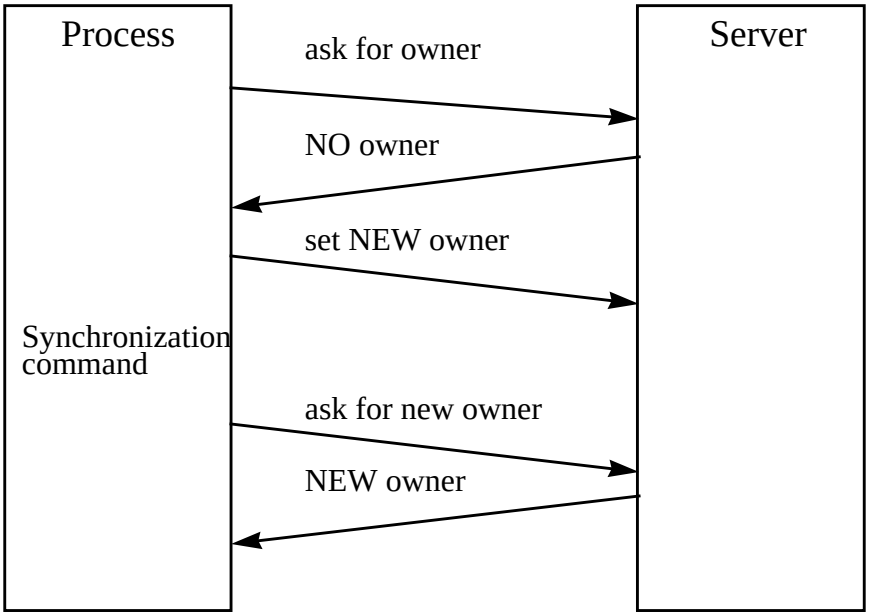

Figure 2. Sequence of events required to lock an area

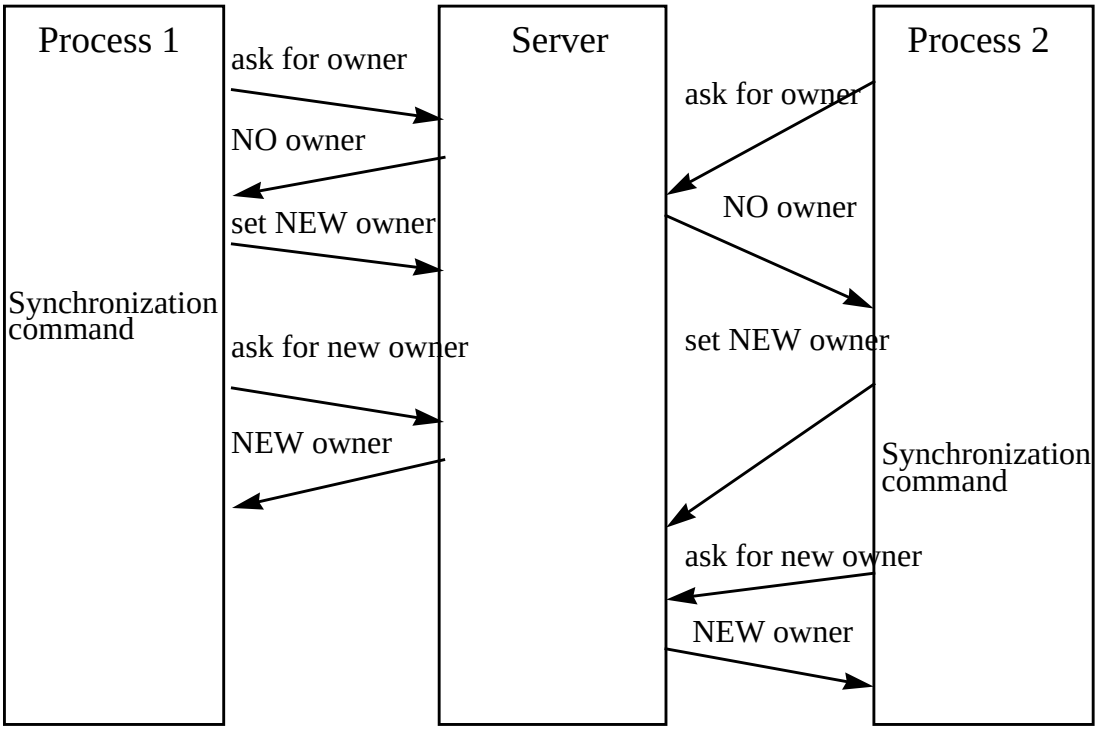

Figure 3. Possible sequence of events leading to lock violation

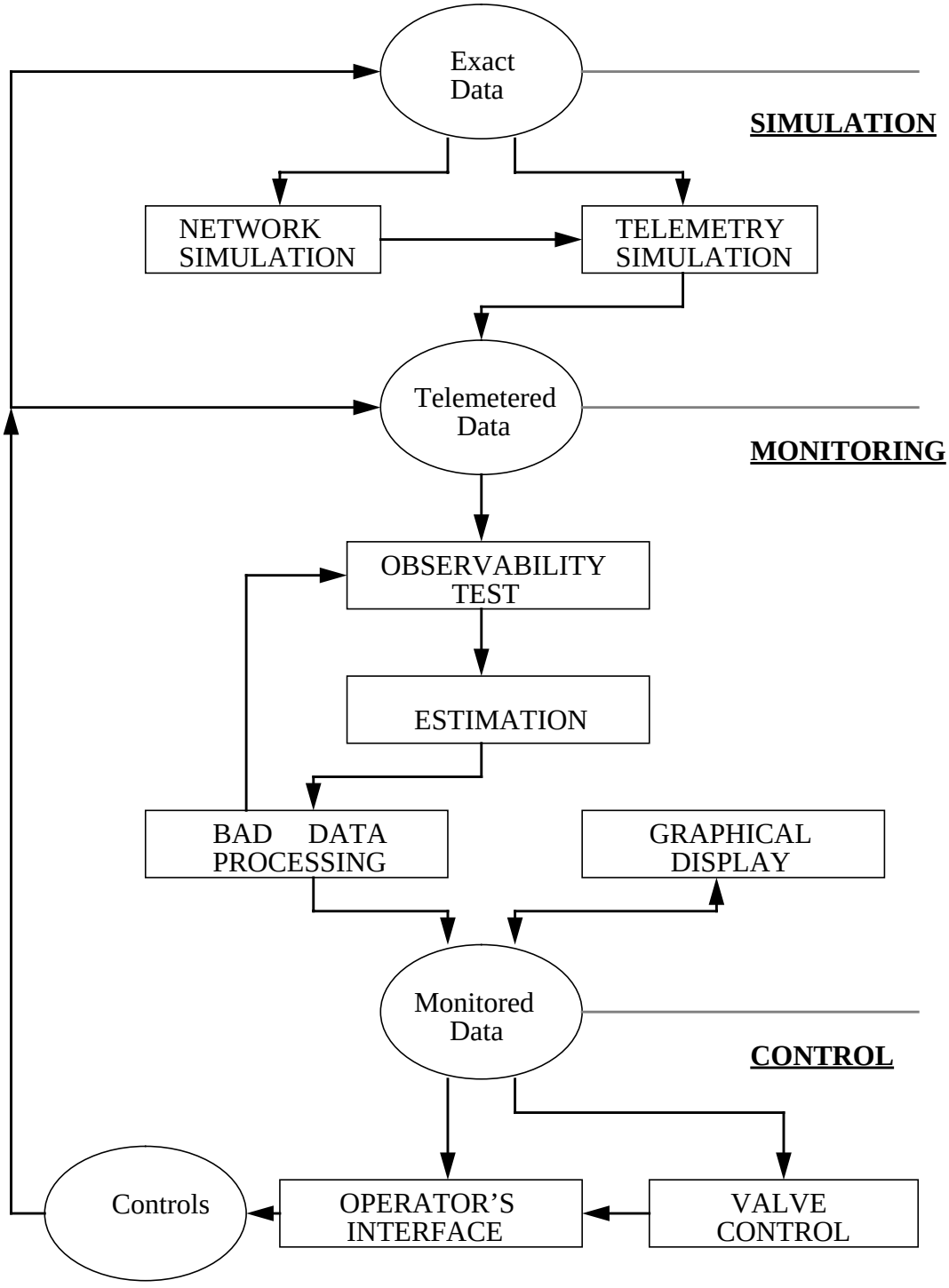

Figure 4. Water network monitoring suite