Check for updates

XDP: A SIMPLE LIBRARY FOR TEACHING A DISTRIBUTED PROGRAMMING MODULE

David M. Arnow
Dept. of Computer and Information Science
Brooklyn College of CUNY
Brooklyn, NY 11210
arnow@sci.brooklyn.cuny.edu

ABSTRACT: *XDP is a simplified interface to the DP distributed programming library. I describe its use in a course on workstation programming, a pragmatic course whose mission is to cover concurrent programming, graphical user interfaces and event-driven programming as well as network and distributed computing. Using XDP, rather than the native socket interface, makes it feasible to cover the last topics, squeezed though they are into a rather overloaded course. Finding (or building) teaching tools like XDP will become increasingly essential as more demands are placed on undergraduate CS curriculum coverage.*

1. So many important topics, so few credits

Until recently, the only exposure to concurrent programming that most undergraduate CS majors received was in the context of an operating systems course. Distributed programming as a topic at the undergraduate level was even rarer. Now, however, it is generally recognized that concurrency, parallel computing and distributed programming are all appropriate and, in fact, highly desirable at the undergraduate level.

There is a problem however. There is already a considerable CS "core" of essential courses, the total number of credits that a college student will take is fixed by the institution and increases in major requirements often meet with stiff resistance from college governance bodies. Furthermore, not only is it increasingly accepted that *all* CS majors have some experience with parallelism and distributed computing but other areas, such as GUIs and human factors, database, optimization, logic programming and so on compete for the all too small set of required advanced undergraduate course credits.

Our response as faculty must be to find parsimonious ways of teaching these areas. We must find vehicles for conveying the important ideas and techniques and avoid spending time on details of minor or ephemeral significance.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGCSE '95 3/95 Nashville, TN USA
© 1995 ACM 0-89791-693-x/95/0003....$3.50

In this paper, I report on the classroom use of a very simple library that supports distributed programming. Section 2 describes the course and explains the need for the library, XDP. Section 3 describes the library itself. Section 4 presents a useful initial classroom example and describes some suitable exercises that use the library. In the last section I draw a few conclusions from this experience.

2. The course: workstation programming

The module appears in a course called "Workstation Programming". It assumes a knowledge of C, some Unix experience and a course in data structures. Its purpose is to introduce students to programming techniques for application development on networks of workstations. It is structured in three modules. At the outset, there is a module that covers process environments, file system issues, concurrent programming, and interprocess communication. It is essentially an overview of basic stand-alone Unix facilities. Another module addresses graphical user interfaces and event-driven programming. A third module deals with network programming: application of transport layer services, remote process creation, distributed computing and the client-server model.

The course has been taught twice and each time has used Stevens' fine book [Stevens90] as a primary text. The students have access to a network of Sun workstations to do their assignments and explore issues raised in the course.

2.1 Distributed and network programming module

This is a pragmatic (dare I say "hacker"?) course and the main goals of the module on distributed and network programming are to give students direct experience with the issues that arise when writing distributed programs that rely on network services. These include but are not limited to

- achieving reliability when the underlying communication medium is unreliable
- synchronization (e.g. barriers, serializing processes)
- parallelizing (or distributing) programs and
- ordering events.

Because the course targets the low-level semantics typically offered by network services, using a high level system such as SR [Andrews82] or Linda [Gelernter85] was in appropriate. The first time I taught the course the students used the BSD socket interface. This approach has many advantages. Our textbook discusses the interface in detail and the interface is ubiquitous on Unix systems. However, like so much of the Unix system, the interface is messy and contains many options and rules that, while perhaps required for generality, are unnecessarily cumbersome for the students. The problem is not so much that the students can't master these rules. It is that the time required for mastery limits the period during which they can be expected to work on interesting exercises. Worse, the interface serves as a distraction and the students often fail to focus on the important issues.

A similar, though less severe, problem had been encountered with the part of the course that used System V IPC features in connection with concurrent programming. That problem had been solved by providing the students with simplified set of routines to create and initialize semaphores as well as provide standard P and V operations.

There are a number of distributed programming libraries available, the most widespread of which is PVM [Sunderam90]. However, even these are too high-level for the purpose needed here. They do not provide messages that cause interrupts nor do they provide messages that are unreliable (in the sense of datagrams). A lower-level library, DP, has been developed at Brooklyn College [Arnow95]. Its semantic level includes that of the socket interface, but it too, again in the interest of generality, has aspects that are distracting to students. For that reason, XDP. a simplified interface to DP, was written and used in the most recent offering of the course.

3. The XDP services

To make use of XDP services in a program, a student need only include the appropriate header file, link to the XDP library and provide a file called "hosts" in the current directory. An XDP computation is initiated by invoking the XDP program, which results in the first, or *primary,* process.

3.1 Process creation and management

Processes are created statically, at the outset of execution. For each entry in the host file, a process will be created, if possible, on the machine described by the entry. In order for this to happen, every XDP program must invoke **xdpinit()**, typically at the outset of execution. When this is executed by the first, or *primary,* process, all the other *secondary* processes are created.

Each secondary process executes the same program as the primary, and thus will also invoke **xdpinit()**. For these processes, **xdpinit()** serves to establish communication. Each secondary process receives the same command-line arguments as the primary. The call to **xdpinit()** returns the actual number of processes (which will be less than or equal to the number of entries in the hosts file).

Each process is identified by a small integer, ranging from 0 to N-1, where N is the number of processes. The **xdpgetpid()** function returns a process's id. The id of the primary will always be 0.

XDP processes must terminate using **xdpexit()**, passing it a string indicating the reason for termination. XDP processes must not use the Unix **exit()**.

3.2 Message sending

XDP processes communicate by sending and receiving messages. To send a message in XDP we write:

```
xdpwrite(id, msgptr, msgsize, mode)
```

where id is an integer that identifies the target, msgptr points to the message data, msgsize is the size of the message, and mode is a set of flags that allow two orthogonal choices:

First, the message can be sent reliably (**XDPREL**) or not (**XDPUNREL**). Reliability here means that XDP will guarantee the eventual sequenced delivery of the message to the target, provided that the underlying network and relevant host machines do not fail. Not sending the message reliably means that XDP will use the cheapest means to send the message and neither sequence nor delivery itself is guaranteed.

The **xdpwrite()** never blocks. This means that upon return from **xdpwrite()**, one thing *is* certain: the target has not yet received the message!

Second, the message can be sent so that it will be "interrupting" (**XDPGETMSG**) or not (**XDPRECV**). In the case of non-interrupting messages, the message is queued upon arrival and does not affect the receiving process until the receiver explicitly reads the message with the **xdprecv()** call (see below). In the case of the interrupting message, the message's arrival may force the invocation of a special message-catching routine if such a routine has been designated by the receiving process. Whether or not such a routine has been designated, the interrupting message must be read explicitly with the **xdpgetmsg()** call, not the **xdprecv()** call.

Receiving messages. Logically, each XDP process has two receiving ports: one for receiving interrupting messages (marked **XDPRECV**) and another for receiving non-interrupting messages (**XDPGETMSG**). XDP processes can receive these messages using one of two routines:

```
xdprecv(src, buffer, limit, mode)
xdpgetmsg(src, buffer, limit)
```

In each case, **src** is a pointer to an integer in which the id of the sender will be stored. The second argument, **buffer** is the address of a buffer in which to store the message contents and the third argument, **limit** specifies its size. Messages that exceed this limit are truncated without remark. The **xdpgetmsg()** call is used to receive messages that

generate an interrupt— as such it is typically used inside an interrupt handler (see below). In such a context, blocking would be inappropriate, and so `xdpgetmsg()` *never* blocks— it returns immediately, with the return value XDPSUCCESS or XDPNOMESSAGE. On the other hand, by using a mode of XDPBLOCK or XDPNOBLOCK, the `xdprecv()` call may or may not be forced to block until a message is available.

Interrupting Messages. In the absence of any arrangement for immediate response to their arrival, interrupting messages can be read in the same way that non-interrupting messages are, but using `xdpgetmsg()` instead of `xdprecv()` (and hence not being able to block). Generally, however, the purpose of such messages is to provoke an immediate response by or have an immediate impact on the recipient. That requires that the receiving process previously invoke `xdpcatchmsg()` (preferably at an early point of the program) passing it a pointer to a message handler, i.e., a function that will be invoked when an interrupting message arrives. The message handler should in turn call `xdpgetmsg()` to retrieve the message and carry out the appropriate action concerning the datum.

In the event that several interrupting messages arrive before the system has had a chance to invoke the message handler function, only one call to the message handler will be made, i.e., there is not a one-to-one correspondence between interrupting messages and calls to the handler. Hence, the message handler should in general be written on the assumption that many messages are ready to be received. As a result, the typical structure of a message handler is:

```
initialization;
while (xdpgetmsg(...)!=XDPNOMESSAGE)
    process message just received;
```

3.3 Synchronization and timeouts

Critical sections. The message handler specified in a call to `xdpcatchmsg()` may be invoked at any time and may reference global objects. To guarantee mutual exclusion, such access should be preceded by a call to `xdpblock()` to disable calls to the interrupt handler and followed by a call to `xdpunblock()` to re-enable them. Upon invoking `xdpunblock()`, if any interrupting messages arrived since the call to `xdpblock()`, the catching function will be invoked.

The message handler is never invoked recursively and so there is no need to protect the function itself. Thus `xdpblock()/xdpunblock()` are never used inside a handler.

Synchronization and timeouts. Sometimes a process needs to wait until some condition becomes true, typically as a result of incoming interrupting messages. In order to do so the process can call `xdppause()`:

```
xdppause(t, f)
```

This call causes the invoking process to suspend execution until an asynchronous event has taken place. Such events include arrival of an interrupting message, a timeout or an external signal. If `t` is not zero, a timeout event is set to occur in `t` milliseconds. In that case, if `f` is not null, it is a function that will be invoked prior to return from `xdppause()`. If `t=0` no timeout event will take place. Since the `xdppause()` routine returns as a result of *any* asynchronous event, interrupting message arrival or otherwise, so the necessary condition must be rechecked:

```
while (!desiredcondition)
    xdppause(t, f);
```

When `xdppause()` is used this way, it is usually the case that the desired condition will become true as a result of the arrival of interrupting messages. There is, therefore, a race condition: after the desired condition is checked, and found to be false, but before `xdppause()` is called, an interrupting message could arrive and bring about the desired condition. Unfortunately, when `xdppause()` would then be called (upon returning from the message handler), there would be no interrupting message to terminate the `xdppause()`— the process would hang. Essentially this is due the global character of the desired condition: testing that condition is a critical section. To avoid this problem, `xdpblock()/xdpunblock` must be used.

```
xdpblock();
while (!desiredcondition)
    xdppause(t, f);
xdpunblock();
```

This code guarantees that the test-and-call sequence (test the condition, call `xdppause()`) is atomic. The message handler is blocked from the time the condition is tested through the time `xdppause()` is entered. In order for `xdppause()` to ever have a chance to complete and to make it possible for the desired condition to become true, `xdppause()` reenables message handler invocations upon enter. Upon return from `xdppause()`, the message handler status is restored to its state upon entry. Note that during `xdppause()`, interrupts are automatically re-enabled.

3.4 Other facilities

To ease the development process somewhat, some utilities for XDP are available:

- `xdphosts` machine1 machine2 ... machineN
  creates a hosts file with the above machines, filling in all necessary fields.
- `xdpsee` program_name
  gives a `ps`-style list of the named program on just those machines mentioned in the local hosts file.
- `xdpcompleted` program_name
  counts the number of XDP processes running the named program on the machines in the local hosts file that have completed.

3.5 Restrictions

Stdin, stdout and stderr are not available to XDP processes: all input and output files must be opened by the processes. Doing a blocking `xdprecv()` is not allowed in a

message handler, nor is sending a **XDPREL|XDPRECV** message. In this section we indicate other restrictions.

Timing. All systems calls and standard subroutines that are implemented using the Unix alarm system call (or its variants) are not allowed. That includes: **sleep**, **alarm**, **ualarm**. To give the application writer some of this functionality, there is a special XDP routine, **xdpalarm(t,f)** which arranges for function f to be invoked after t milliseconds.

Asynchronous and signal-driven i/o. Using the BSD **select()** system call or making use of the SIGIO signal is forbidden.

Exec. Use of any of the **exec** variants is forbidden, unless used in conjunction with **fork()**.

Fork. The **fork()** system call can be used provided that the children to not attempt to partake in the execution of xdp routines. Child processes (but not the parent) may do execs.

4. Examples and sample problems for students

An straightforward example for illustrating some of the issues in distributed programming is the following program, wl 'ch creates a file containing all the primes from 2 to a number that is a command-line argument to the program. The range of integers to be checked is divided up into equal subranges. The primary informs all the secondary processes what their subranges are. (The primary has a subrange for itself too.) Each process works independently for the most part, checking integers in its subrange (executing **search()**. When the primary finds a prime it calls **newprime()** to add it to the list. When a secondary finds a prime it sends it to the primary using **xdpwrite()**.

The latter information INTERRUPTS the primary, forcing the invocation of **fcatch()**. This in turn invokes **newprime()**. Termination handling is done as follows: secondaries send a negative integer to the primary to indicate they are finished. When the primary has received the appropriate number of such integers it knows the computation has ended.

```
#include        <stdio.h>
#include        "xdp.h"

#define   REL_RECV  (XDPREL|XDPRECV)
#define   REL_INTR  (XDPREL|XDPGETMSG)

#define MAXPRIMES       90000
int  p[MAXPRIMES], nprimes=0;

int  mypid, nnodes, done=0, interval, maxnum;

void                        /* executed by the primary only */
newprime(int n) {
     xdpblock();            /* potential race condition */
     if (nprimes<MAXPRIMES) /*  so block interrupts   */
          p[nprimes++] = n;
     xdpunblock();
}

void
sendint(int dest, int v, int mode) {
     xdpwrite(dest, v, sizeof(i), REL_INTR);
}

void           /* executed by the primary only */
fcatch() {
     int     p, src;

     while (xdpgetmsg(&src,&p,sizeof(p))
                         !=XDPNOMESSAGE)
          if (p < 0)
               done++;
          else
               newprime(p);
}

void
search(int n1,int n2) {
     int     i;

     for (i=n1; i<=n2; i++) {
          if (IsPrime(i) {
               if (mypid==0)
                    newprime(i);
               else
                    sendint(0,i,REL_INTR);
          }
     }
}

void
foreman() {    /* executed by the primary only      */
     int     i, pid, b;

     xdpcatchmsg(fcatch);

     for (b=0, pid=1; pid<nnodes; pid++) {
          sendint(pid,b, REL_RECV);
          b += interval;
     }
     search(b,maxnum);
     done++;

     xdpblock();
     while (done<nnodes)
          xdppause(0L, NULLFUNC);
     xdpunblock();

      ... print the list of primes here ...

     xdpexit("You're fired!");
}

void
worker() {     /* executed by the secondary only     */
     int     b, src;

     xdprecv(&src, &b, sizeof(b), XDPBLOCK);
     search(b, b+interval);
     quitvalue = -1;
     sendint(0,-1,REL_INTR);
     fclose(fp);
     xdpexit("I quit!");
}

main(int ac,char *av[]) {
     nnodes=xdpinit(&ac,&a);
     maxnum = atoi(av[1]);
     mypid  = xdpgetpid();
     interval = maxnum/nnodes;
     (mypid == 0) ? foreman() : worker();
}
```

Along with examples like this students have been given various exercises, such as:

- parallelize a simple quadrature
- parallelize (Djikstra's) single source shortest path algorithm
- parallelize a given ising-model simulation
- write a program that estimates the relative available computational power on the set of machines mentioned in the hosts file
- using `xdpalarm()` arrange to send messages reliably using only the `XDPUNREL` flag in `xdpwrite()`

5. Conclusion

As can be seen from the above, the student working with this library still must address the fundamental problems of buffering, race conditions, synchronization, task decomposition, and reliability that must be resolved in network computing. By using the library, the student is relieved of having to create and bind addresses and carry out other messy socket layer administration.

XDP is a new library and, so far, has not been used outside of Brooklyn College. Is this a disadvantage for these students? In the absence of time constraints, using the socket layer might be desirable, but even if there is enough time to teach it effectively, should a CS major's first encounter with network computing be obscured by its idiosyncrasies? And would a student graduating to a position involving network programming under NT be better prepared by an XDP or a socket layer module? My admittedly anecdotal (based on two offerings) experience suggests that using a library to raise just the issues of interest and to hide those not of interest is effective. Others in my department have had similar experiences in teaching other advanced undergraduate electives, for example, constraint-based programming [McAloon,Tretkoff95]. As we in the university strive to make available a broader range of more advanced topics, we may increasingly have to resort to libraries and environments that are specially designed for instruction.

6. References

[Andrews82] G.R. Andrews: The distributed programming language SR-- mechanisms, design and implementation. *Software— Practice and Experience 12*,8 (Aug. 1982).

[Arnow95] D.M. Arnow: DP: A library for building portable, reliable distributed applications. *To appear in the Proceedings of the Winter USENIX 95 Conference*, New Orleans (Jan., 1995).

[Gelernter85] D. Gelernter: Generative communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (Jan. 1985).

[McAloon,Tretkoff95] K. McAloon and C. Tretkoff: *Optimization and Computational Logic*. Wiley, to appear in 1995.

[Stevens90] W. Richard Stevens: *UNIX Network Programming*. Prentice-Hall (1990).

[Sunderam90] V.S. Sunderam: PVM— A framework for parallel distributed computing. *Concurrency: Practice and Experience 2* (1990).