# DP: A library for building portable, reliable distributed applications

*David M. Arnow*
*Brooklyn College, CUNY*

**ABSTRACT**

DP is a library of process management and communication tools for writing portable, reliable distributed applications. It provides support for a flexible set of message operations as well as process creation and management. It has been successfully used in developing distributed Monte Carlo, disjunctive programming and integer goal programming codes.It differs from PVM and similar libraries in its support for lightweight, unreliable messages, as well as asynchronous delivery of interrupt-generating messages. In addition, DP supports the development of long-running distributed applications tolerant to the failure or loss of a subset of its processors.

## 1. Distributed Programming Tools

Although the increase in diversity and availability of parallel multiprocessors shows no sign of abatement, the one truly ubiquitous parallel computer system continues to be the LAN of workstations and the one massively parallel system to which "everyone" has access, if not authorization, is the Internet. Recognition of network computing as an important platform for parallel computing and the desirability of high-level and portable programming systems has resulted in the widespread development of a host of message-passing based programming environments.

Initially, much of this effort went into the design of programming languages or language extensions. An extensive review of these is given by Bal [Bal89]. Each language, besides providing a higher semantic level and portability expresses a view as to how a distributed program ought to be conceived. These views may be limiting. For example, many languages (e.g. SR [Andrews82]) are strongly influenced by the semantic restrictions (synchronous message passing) advocated by Hoare [Hoare78]. Other languages adopt an asynchronous message semantics (for example, NIL [Strom83]). Still others hide message passing altogether, and present a paradigm different from that of a distributed system. Most notable among these are parallel logic programming languages such as PARLOG [Clark88], the shared memory model of LINDA [Gelernter85], or more recently Concert/C [Aurebach92; Goldberg93].

Languages for distributed systems are necessarily designed with a particular paradigm in mind and as such must impose some restrictions in order to maintain the integrity of that paradigm. The portability of their implementation is not trivial. Perhaps even most significantly, new languages require a substantial reinvestment on the part of users. Therefore, as the locus of interest in parallel network computing changed from language designers to users, there has been a shift to the design of libraries of standard routines or of environments consisting of supporting processes as well as libraries. While the programmer is no longer protected by a language, a greater flexibility and portability can be achieved.

One such environment is PVM [Sunderam90; Geist92], which is implemented on a variety of Unix systems and enjoys extensive use by computational scientists. Others include NMP [Marsland91] and P4 [Butler92]. The performance of these systems and others has recently been reviewed in two papers [Douglas93 and Parsons94]. In response to both the proliferation of such environments and the use of PVM as a de facto standard, the past two years have seen an effort to develop a standard for these environments, MPI [MPI93].

All of these environments provide varying degrees of flexibility, portability and scalability, with PVM providing the most. However, none of them offer the flexibility that my applications required.

Furthermore, none offer the kind of reliability that is necessary for conveniently scaling up to long computations involving many workstations.

## 2. Wanted: Portability, Flexibility and Reliability

**Portability**. DP was developed as a result of my own experiences writing distributed programs that ran on LANs and the Internet itself from 1988-1991. These programs included Monte Carlo and other scientific calculations as well as operations research programs. Writing the process management and communication code directly in the native system primitives was maddeningly non-portable even though all the systems involved were either some flavor of Unix or inspired by Unix. The programs were parallelizations of large existing codes, and the necessary interprocess communication was embedded deeply so rewriting these programs in a different language was out of the question.

**Flexibility**. Most frustrating was the loss of flexibility (with respect to use of the native system primitives) that results from the use of any of the other distributed languages or programming environments available then and now. The communication facilities available in these systems (and described in the proposed MPI standard) do not support *interrupting* messages. Thus, a process receiving a message must invoke a receive operation explicitly at each point that a message is sought. The receive operation is typically allowed to be blocking or non-blocking, so both barriers and polling are readily available to the programmer.

In situations where a process cannot proceed at all until data from an incoming message has been received, these message semantics pose no problem— it is entirely natural to explicitly encode message receive operations just prior to the use of the needed data in the program and entirely proper for these operations to be blocking.

However, there are situations, for example in the Monte Carlo and in disjunctive programming applications in which the I was interested, where:

(a) incoming data serves "merely" to increase the efficiency of the processes computation

and

(b) it is not certain that the incoming data will arrive at all!

In these situations, it is both unnatural and extremely inefficient to explicitly encode non-blocking receive operations in the process's application code. In some cases, the problem is mitigated by the availability of threads. A dedicated input thread can integrate the contents of an incoming message into the process's data objects without the need for polling. However, it may be that the only way to efficiently respond to the new information is for the main thread to make an abrupt change in its control, i.e. to make an sudden jump out of its current nested routine stack. In the absence of an inter-thread signaling facility, there is no way for the main thread to recognize the need for this short of testing an object in its own address space— cheaper than executing non-blocking receives, but still inefficient.

Another loss of flexibility is the inability to send fast UDP-style messages in situations where message unreliability may not be a serious drawback. Operating system services often have this characteristic, but, surprisingly perhaps, so do some applications. Consider, for example, a large Monte Carlo calculation involving thousands of random walks. It is often the case that if a small fraction of these are, at random, lost (as a result of message loss), then the impact is "only" an increase in variance. If as a result of permitting such losses, the computation can run faster and hence have a greater sample size, then the benefit could outweigh the loss.

**Reliability— for the sake of scalability**. Although reliability was decidedly *not* an initial concern of this project, the project's own success forced the issue. The applications that most readily make use of DP are those which have a high computational cost and which are parallelizable. But by running a parallelized application over a great number of workstations on a LAN for a long time, the likelihood of zero workstation reboots during the course of a single computation began to become uncomfortably low. In order to more fully realize the potential for the exploitation of networks and internetworks of workstations, greater reliability is essential. So, as the project developed, increased reliability (with respect to single workstation failure) became a goal.

In summary, the DP library was designed with the following goals:

(1) *Flexibility and power*: The primitives must provide the power to perform most distributed programming functions. The application programmer should not lose any functionality or efficiency by using DP instead of the native system primitives.

(2) *Portability*: The primitives should be implementable on most, if not all, distributed computing platforms.

(3) *Reliability:* the loss due to external circumstances of one or all processes on a single workstation should have no impact on the outcome of a distributed computation other than a short delay, provided that the processes involved do not conduct any i/o other than message sending and receiving.

With the exception of not providing a broadcast or multicast message facility, DP meets these goals. On the other hand, in comparison to other distributed programming environments and languages, DP provides a very low-level application interface. There is no typing of messages, minimal support of data format conversions, no queuing of synchronous messages, and no concept of conditional receives. There is, however, a higher-level distributed programming support environments, stdDP [Arnow94] that provides those services and is implemented using DP.

## 2.1 A sketch of a motivating application: capacitiated warehouse location

To clarify the kind of problem that demands the interrupting message facility that is absent from other environments, we present one example: the capacitated warehouse location problem— a classic operations research problem.

The goal is to supply the needs of customers from a group of proposed warehouses and to minimize monthly cost. Using a warehouse requires a fixed monthly charge and there is the cost of supplying all or part of a customer's requirements from a given warehouse. The problem is to determine which warehouses to open so as to minimize cost while meeting demand. Although the problem is NP-hard, good results can be achieved using Branch-and-Cut and Branch-and-Bound techniques. Worker processes are given portions of the search tree to explore and communicate intermediate results to another. Idle worker processes are given new tasks by master processes which, must obtain these from busy worker processes. Worker processes can complete their tasks significantly more rapidly through pruning by "knowing" the current global minimal cost.

Both of these operations— obtaining the new global minimum and paring the current subtree to define and transmit a new task— are in response to arriving messages, the number of which is unknown. Polling, besides being inefficient requires an inordinate modification of the original code. In both of these cases, efficiency and convenience is served by providing interrupting messages. This application and other related ones are described in [Arnow91, Arnow94, Arnow95].

# 3. DP's services

Although of the same genus as environments such as PVM and P4, DP differs from each of them in a number of important ways, primarily because of the above goals. Functionally, the most important difference is its provision for unsolicited messages whose arrival generate a software interrupt. This provides a flexible method of sending large quantities of urgent information that cannot be easily accomplished with, say, the unix-signal transmission of PVM. It also allows a programming environment to provide a shared-memory like capability. DP permits messages to be sent in the cheapest way possible, when reliable transmission is not necessary. Messages can be received with or without blocking. Process creation is dynamic and limited only by available computing resources. Furthermore, DP can guarantee the reliability of those processes that engage in no i/o other than message sending and receiving in the event of a single workstation failure. This section presents an overview of most of the services provided by DP.

## 3.1 Process management

**Execution**. Execution of a DP application starts by invoking a suitably compiled and linked DP executable program. In DP parlance, this process is called *the primary*, though its primacy is for the most part just a matter of being first— there is nothing very special about the primary. The primary process and its descendants (those process that are spawned by it or its descendants) constitute a *DP process group*. DP processes can only communicate within this group.

**Identification**. Each DP process is identified by a value of type `DPID`, guaranteed to be unique among *all* possible DP processes. The function **`dpgetpid()`** returns the current process's id via a storeback parameter. Processes learn the DPIDs of other processes either by being their parent, by receiving a message from them, or when the DPID of a process is in the contents of a message and is used by the receiving process as such.

**The hosts file.** In order to spawn processes, a DP program must have information about the available hosts for process creation. DP processes can acquire that information dynamically but it is usually convenient to provide that information to the primary via a *hosts file* in the directory from which the DP application is executed. The primary will automatically read this file and prompt the user for the passwords needed to access the networks named. This information is inherited by spawned processes.

**The hosts table.** Host information is maintained internally in a *hosts table*. In DP application code, hosts are identified using integer indices to this table. The table is dynamic— new hosts can be introduced during run-time by the function **`dpaddhost()`**. This call is an alternative and a supplement to providing host information through a static, though convenient, hosts file.

**Process creation.** Process are spawned by calling **`dpspawn()`**, passing

- the name of the program to be executed,
- the integer index of the host on which to run the new process,
- a *semantic packet* to be sent to the new process; this packet is a program-determined collection of bytes that can be used as an initial parent-to-child communication— typically it contains the DPID of the parent, along with possibly other application parameters.

The call to **`dpspawn()`** returns the DPID of the new process via a storeback parameter. The entry point for the newly spawned *secondary* processes is **`main()`**, not the instruction after the call to **`dpspawn()`**. These secondary processes do not have access to the original command-line arguments nor do they inherit a copy of the creator's address space— their data must come from the semantic packet or from subsequent received messages. Guiding **`dpspawn`** is the entry in the internal host table for the given host id. That entry, among other things, determines the user name under which the new process will run and most significantly the directory in which the program must be found and in which the new process will start executing

**Initialization.** The first DP call any DP program makes should be **`dpinit()`**, which sets up the necessary process and communication environment. This includes initialization of DP's data structures and establishing an address. If the process calling **`dpinit()`** was created by **`dpspawn()`** the caller is given access to the *semantic packet,* described above. A pointer to this packet is returned via storeback parameter to **`dpinit()`**. The size of the packet is also stored back. In the case of a primary process, there is no semantic packet and the size stored back is –1: that is how the code can determine whether it is running in the primary process or a secondary after a call to **`dpinit()`**.

The call to **`dpinit()`** completes the handshaking with the creating parent process. The creating process cannot continue its work until the created process makes this call. For this reason, the call to **`dpinit()`** should be made as soon as possible. Upon returning from **`dpinit()`**, the process is a genuine DP process and can partake in the activities of the DP family.

In all cases, **`dpinit()`** returns the number of host machines in its inherited internal host table and stores back the host id of the machine on which the process is running.

**Joining a DP process group**. Any non-DP process may join an existing DP process group. For this to be possible, one or more of the processes in the group must invoke **`dpinvite()`**. This call creates a *contact file*, which contains all the information that a new process would normally get from **`dpspawn()`**. All the joining process need do is invoke **`dpjoin()`** with the pathname of the contact file as an argument. This call plays the role of **`dpinit()`** and establishes communication using information provided in the contact file. The newly joined DP process's identity can then be conveyed to any process in the group. Note that this mechanism requires that the joining process and the inviting process must share some file address space in common.

**Finishing Up.** All DP processes must call **`dpexit()`** to make a graceful exit. The **`dpexit()`** function is the DP substitute for Unix **`exit()`** call; that is, it makes a no-return exit. If a DP process fails to exit using **`dpexit()`**, i.e. if it exits using the Unix **`exit()`**, other DP processes in the application may fail. The main purpose of **`dpexit()`** is to withdraw the exiting process from contact with the remaining DP processes prior to an actual exit in a way that guarantees correct message transmission. The only argument to the function is a string identifying the reason for termination. The string appears only in the log file for the process and may be null.

Sometimes, it may be desirable for a process to cease DP activity but persist in some other activity. By passing the address of a function to **`dpsetexfun()`** any time prior to calling **`dpexit()`**, a process guarantees that **dpexit()**, after withdrawing from the group of DP processes, will call the indicated function prior to doing the actual Unix exit.

**Bailing out.** The **dpexit()** call terminates one DP process in the group. Generally, each process's own logic dictates when that termination is appropriate. In exceptional circumstances, it may be necessary to allow a single process in the group to force termination in the entire group. In such a case, **dpstop()** can be called. The **dpstop()** call force immediate shutdown of all processes. The function set by **dpsetexfun()** is *not* called and the ensuing shutdown is so radical that even earlier messages that had been sent but were not yet delivered may be thrown away. The function receives one argument, a string, which has the same meaning as the string passed to **dpexit()**.

## 3.2 Communication

**Sending messages.** DP processes communicate by sending and receiving messages. For sending messages, the **dpwrite()** routine requires the DPID of the recipient and a pointer to the start of the message body along with the message body size. A variant, **dpsend()**, allows a message body to be specified as a linked list. Messages can be reliable or non-reliable and interrupting or non-interrupting. Reliability here means that DP, which as I describe in section 4 uses UDP as its underlying protocol, will carry out an ack/timeout/retransmit protocol that will guarantee the eventual availability of the message to the target provided that the underlying network and relevant host machines do not fail. Reliable messages are received in the order in which they were sent. Sending the message unreliably means that DP will send the message to the target only once and assume no further responsibility— a much cheaper method of message transmission.

Regardless of whether the message is sent reliably, return to the sender is immediate; the sending process will not be blocked during this time. So upon return from **dpwrite()**, one thing *is* certain: the target has not yet received the message.

**Receiving messages.** Logically, each DP process has two receiving ports: one for receiving interrupting messages and another for receiving non-interrupting messages. Non-interrupting messages are queued upon arrival and do not affect the receiving process until it explicitly reads the message with the **dprecv()** call. 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 via a call to **dpcatchmsg()**. Whether or not such a routine has been designated, the interrupting message must be read explicitly with the **dpgetmsg()** call, not the **dprecv()** call. Both routines return the DPID of the sender as well as the message itself and both routines move the incoming message from an internal DP buffer to an application-provided buffer. If the latter is insufficient to hold the message, the message is truncated. The **dprecv()** call can be made with or without blocking semantics, but the **dpgetmsg()** call, because it is typically used inside an interrupt handler where blocking would be inappropriate *never* blocks. 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 must be assume that there may be more than one interrupting message ready to be received.

**Longjumps.** Sometimes when the message-catching routine is invoked, it responds to the incoming information by modifying a global data structure or sending out a message with requested information. At other times, however, it must respond by making an *exceptional* change in the control flow of the receiving process. The **dplongjmp()** routine provides that capability. It works exactly as **longjmp()** does and in fact its argument is a jmpbuf that was set by **setjmp()** (there is no "dpsetjmp"). The only reason for **dplongjmp()** (instead of the standard **longjmp()**) is that the jump *out* of the message handler must be accompanied by a re-enabling of interrupting messages.

## 3.3 Synchronization and timeouts

**Critical sections.** The application-specified message-catching routine may be invoked at any time and may reference global objects. Thus, any other code that accesses these global objects is a one-way critical section, in the sense that though, upon receipt of an interrupting message control may transfer from the critical section to the handler, the reverse is not possible: control will not pass from the handler until it has completed its work and returns. To guarantee mutual exclusion, such access should be preceded by a call to **dpblock()** to disable calls to the interrupt handler and followed by a call to **dpunblock()** to re-enable them. Upon invoking **dpunblock()**, if any interrupting messages arrived since the call to **dpblock()**, the catching function will be invoked.

**Synchronization and Timeouts.** Sometimes a process needs to wait until some condition becomes true, typically as a result of incoming interrupting messages. The **dppause()** call suspends execution of the process until any asynchronous event takes place. The application may set a timer and a timeout function through non-zero arguments to this call. Upon entering **dppause()**, interrupting messages (and calls to the message catcher) are enabled and status is restored upon return. Typical use of this function is

```
dpblock();
while (!some_desired_condition)
    dppause(0, (FUNCPTR) 0)
dpunblock();
```

The intent of this code is not to proceed until **some_desired_condition**, which presumably depends on the arrival of a message, is true. Rather than busy-wait, the program calls **dppause()** which will not return until some event, possibly a message arrival, has taken place. Because many events are possible, the desired condition has to be rechecked and dppause() reentered if necessary. The window between the checking of the desired condition and entry into **dppause()** open the possibility for a race condition and so the loop is enclosed by calls to **dpblock()** and **dpunblock()**.

### 3.4 Restrictions and Application Front Ends

Except for processes that use **dpjoin()** to join a DP process group, standard input/output/error are not available to the DP application. Thus **dpjoin()** is essential if interactive programs are desired. Message-catching functions may not call **dprecv()** in blocking mode.

**Timing.** All systems calls and standard subroutines that are implemented using the Unix alarm system call (or its variants) are not allowed because they would interfere with DP's own reliance on this facility. That includes: **sleep**, **alarm**, **ualarm**. To restore some of this functionality to the application writer, there is a special DP routine, **dpalarm(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 and fork**. Use of any of the **exec** variants is forbidden, unless used in conjunction with **fork()** or after **dpexit()** has been called. The **fork()** system call can be used provided that the children do not attempt to partake in the execution of DP routines. Child processes (but not the parent) may do execs.

**Application front ends**. These restrictions might initially seem daunting to the application writer. However, it is always possible for non-DP processes, such as one intended to support an event-driven user interface front end, to fork a child process which uses **dpjoin()** to become a DP process or even which uses **dpinit()** to become a DP primary process. The non-DP parent and the DP child can communicate using pipes or SysV IPC.

## 4. Examples

**A simple example: primes.** The primes program, shown below, illustrates the use of the DP interface. The primary process uses **dpcatchmsg()** to arrange for **fcatch()** to be invoked in the event of an interrupting message and then spawns two processes for every available host, sending a semantic packet containing just the DPID of the primary to each secondary process. It divides the interval 1..100000 equally among all the processes, including itself and then uses **dpwrite()** to send the lower bound of each subinterval to each process in a reliable, non-interrupting message (**DPREL|DPRECV**). The primary then searches for primes in its own subinterval.

Meanwhile, the secondary processes have started and, having received their lower bound by calling **dprecv()**, they too start searching for primes in their own subintervals. Both secondary and primary processes invoke **newprime()** when a prime number is found. For the primary, newprime just adds the prime to the set of primes— this is a critical section because an interruping message may access the same set and so must be protected with **dpblock()** and **dpunblock()**. For the secondaries, **dpwrite()** is used to send the prime number in a reliable, interrupting message (**DPREL|DPGETMSG**). The arrival of these message cause **fcatch()** to be invoked, and the incoming prime number to be stored in the set of primes.

To let the primary know that no more primes are forthcoming, secondaries send a negative integer in a reliable interrupting message and then exit. The primary waits till it has received the appropriate number of such messages and then exits.

```
#include <stdio.h>
#include <dp/dp.h>

struct semstr {        /* most programs would have */
     DPID s_id;        /* other fields here as well  */
} s, *sp;

#define   MAXPRIMES 100000
int  p[MAXPRIMES], np=0, IsPrimary,
     nprocs, nhosts, done=0,
     interval, myhostid;

#defineRelInt(DPREL|DPGETMSG)
#defineRelNonInt(DPREL|DPRECV)

void
sendint(DPID *dest, int i, int mode) {
     dpwrite(dest, &i, sizeof(i), mode);
}

void
newprime(int n) {
     if (IsPrimary) {
          dpblock();    /* potential race condition */
          p[np++] = n;/*   so block interrupts    */
          dpunblock();
     } else
          sendint(sp->s_id, n, RelInt);
}

void
fcatch() {
     int  v;
     DPID src;

     while (dpgetmsg(&src,&v,sizeof(p)
                    !=DPNOMESSAGE)
          if (v<0)
               done++;
          else
               p[np++] = v;
}

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

     for (i=n1; i<=n2; i++)
          if (IsPrime(i))
               newprime(i);
}

void
primary(char *prog) {
     int  i=1, v=0;
     DPID child;
     FILE *fp;

     dpcatchmsg(fcatch);
     dpgetpid(&s.s_id);
     while (i<nprocs) {
          dpspawn(prog, &child, i%nhosts,
                         &s, sizeof(s));
          sendint(&child, v, RelNonInt);
          1+=interval
          i++;
     }
     search(v,MAXPRIMES);
     done++;

     dpblock();        /* potential race condition: so  */
     while (done<nprocs) /*  block interrupts       */
               dppause(0L, NULLFUNC);
     dpunblock();
                         /* write primes to results */
     output("results", p, np);
     dpexit("You're fired!");
}

void
secondary() {
     int  v;
     DPID src;

     dprecv(&src, &v, sizeof(v), DPBLOCK);
     search(v,v+interval);
     sendint(sp->s_id, -, RelInt);
     dpexit("I quit!");
}
                    /* main: executed by all processes*/
main(int ac, char *av[]) {
     nhosts = dpinit(av[0], &sp,
                         &size, &myhostid);
     IsPrimary = size==(-1);
     nprocs = 2*nhosts;/* 2 processes per host*/
     interval = MAXPRIMES/(nprocs+1);
     IsPrimary? primary() : secondary();
}
```

**Capacitated warehouse location problem, again.** A branch-and-bound search for solutions can be efficently parallelized using DP. N processes are created, N being determined by available hardware. The primary maintains a set of unsearched subtrees— initially this set is take from the top N subtrees of the search tree. When a secondary process becomes idle, it sends an reliable interrupting message to the primary requesting a subtree and then waits until it recieves one. When the primary's set of unsearched subtrees falls below a low-water mark, it sends reliable interrupting messages to all the active secondaries, requesting that they split their subtree at the next convenient point. These will continue to split their subtrees, sending (in reliable interrupting messages) the split-off branches to the primary to replenish its set, until the primary, having passed a high-water mark, sends them reliable interrupting messages to desist. This is very effective load-balancing. The availability of interrupting messages here is essential because of the unpredicatibility of need and availability of search subtrees on the one hand and the undesirability of frequent polling on the other.

An important element of the branch-and-bound search algorithm is the ability to prune search subtrees when the best extremum the subtree can offer is inferior to the best extremum already encountered. In a shared memory environment, all processes have memory access to the best extremum but in a message-passing network environment making sure this information is rapidly available to all processes is both necessary and non-trivial. Using DP, this problem is efficiently addressed as follows. Whenever a secondary discovers what appears to be a new best extremum it sends a reliable interrupting message to the primary, which multicasts this in lightweight (unreliable) messages to all the other secondaries. Making these message interrupting guarantees that the information will become available to the receiving process as quickly as the underlying system permits. Using lightweight messages ensures that the multicasts will not overload the system nor overbuden the primary. The cost of occasionally losing such a message is minor: it simply means that occasionally for some, usually small, duration, a secondary may not be pruning its subtrees as effectively as it would otherwise.

## 5. Implementation

DP is implemented using the socket system call interface to the TCP and UDP services of the TCP/IP protocol suite, basic Internet services such as ping and rexec and, of course, a host of Unix services. Once the basic implementation issues were decided all of these services were used in the obvious way.

**Communication mechanism.** The first issue to be decided was how inter-process communication is to be handled. PVM and many other environments use TCP. This is a very attractive choice given that much inter-process communication has to be reliable and that TCP handles this within the OS kernel. Building a reliable service using UDP requires duplicating much of this outside the kernel with all the context-switching cost that this implies. Nevertheless, DP's inter-process communication is almost entirely implemented using UDP. The reasons for this are:

- TCP requires maintenance of connections and Unix (and presumably most systems) place limits on the number of connections that a process can maintain. The choice then is to take down and recreate connections as needed (too expensive), limit the number of processes with which a process can communicate (clearly unacceptable), or implement a routing mechanism. This would have to be done outside the kernel, negating in part the purpose of using TCP in the first place, especially when large numbers of processes are involved.
- Efficient as TCP is, sending a UDP packet into the ether is cheaper, and because the design of DP was predicated on the desirability of low-cost unreliable messages in many cases, it seemed a shame to pay more than necessary for that kind of communication.
- The fault-tolerant mechanism, described below, is greatly simplified using UDP rather than TCP. This was *not* a reason at the outset of this project for using UDP, but it became a reason for being very glad about the choice!

Using UDP does require DP to guarantee reliable sequenced delivery of those messages that require this service. In the current implementation, reliable messages are implemented in the most naive way (with sequence numbers, positive acks, timeouts, retransmits and a notion of "stale" messages). More sophisticated implementations are certainly possible and in the still "gray" part of the DP interface, there are calls that allow the programmer to adjust protocol parameters such as timeout size.

**Process identification**. It would have been desirable for DPIDs to be integers or some other basic C type. However, that requires some kind of id-to-address mapping internal to DP. A problem arises when an application DP process references a DPID for which its own DP runtime support does not have a mapping. This could and does arise when DPIDs are sent in application messages. To eliminate any need for a centralized or distributed id resolution mechanism, DPIDs are *not* integers but 28-byte structures containing all the information needed to address the corresponding process and more. The additional information represents “the kitchen sink”. Some of it, in retrospect, has turned out to be useful— other components (indicating what protocol— for example IPX— is involved) may never find a use.

There are parallel methods which naturally assign integers to a set of processes and use these assignments in their algorithms. As it turns out, the use of a non-basic type does not pose much of a problem in those cases. Programs which use such methods do *not* spawn processes dynamically but rather use a fixed number of processes created from the outset. The need to create an initial DPID-to-integer map is only a minor inconvenience to the application writer, and there are libraries, such as stdDP [Arnow, 94] built on top of DP that provide this service, along with others. From an esthetic point of view the chief regret with this choice is the necessity for providing a **`dpidmatch()`** function. On the other hand, from an implementation point of view, things are greatly simplified.

**Process creation and initialization.** Processes are created using the rexec service. In order to spawn a process, the creator, after checking the argument to **`dpspawn()`** as best as possible, forks a child process which does most of the work. The child process calls rexec and uses the resulting TCP connection to deliver the semantic packet and the internal hosts table to the newly spawned process. It uses the same connection to receive the new process's DPID (which contains, among other data, its UDP address). This is the only use of TCP in the implementation. The parent receives the DPID from the child through a pipe and waits for the child process to complete the handshaking with the new process and disappear, along with its TCP connection. This arrangement avoids the need for a separate call by the parent to recognize the completion of the process creation. Although it compels the parent to wait until the new process is created, the creator is still able to receive and respond to interrupting messages— an allowance which is made much easier by having a child process do most of the work.

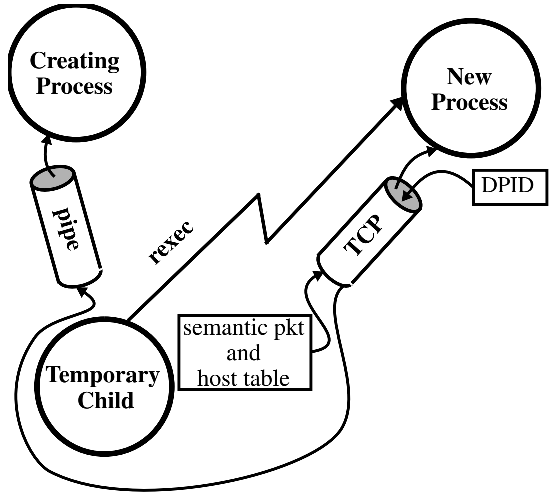



**Figure 1: DP process creation**

*A temporary child process creates the new process using rexec. The resulting TCP connection is used tosend the semantic packet and receive the DPID.*

**Reliability.** The scheme for enhancing reliability is inspired by one used, in a different context, in the early 1980s in the design of a fault-tolerant Unix box based on a shared memory architecture [Borg83]. Each active process is created with a backup process residing on a different workstation. The workstation housing a backup must be binary compatible with that executing the active process. Furthermore, because of the way recovery is implemented, each pair of active and backup processes must share some file address space in common (though between two distinct pairs there is no such need). The scheme only *guarantees* against single workstation failure, though it *may* work in the event of multiple failures. What it requires is that the workstation holding the backup process not fail. So in the figure below, had th backup of process C been executing on an additional machine, say Sparc#4, rather than on Sparc#2, then both Sparc#1 and Sparc#2 could have failed simultaneously.

When process A sends a message to process B, process B uses the message and sends a copy of the message to its backup, B'. If the message is a reliable message, B' sends the acknowledgment to A. Thus A continues to retransmit in the usual way until it is certain that both B and B' have the message. Redundant transmits to B cause no problem because they are simply stale messages which an ack/timeout/retransmit protocol would ignore anyway. B' saves all messages (until, as described below, a checkpoint operation takes place). Upon detection of failure (see below), B' starts executing. Since it has all the messages (with the possible exception of a few unreliables) that B received, its execution will be identical. It will send output messages that are redundant to ones sent by B previously, but these will be treated as stale by their recipients and not cause any inconsistency.

**Figure 2: Backing up processes**

*Each process is created with a backup. The backup process must be on a different , but binary-compatible machine. Ideally, the distribution of backups and processes is consistent with some notion of load-balancing.*

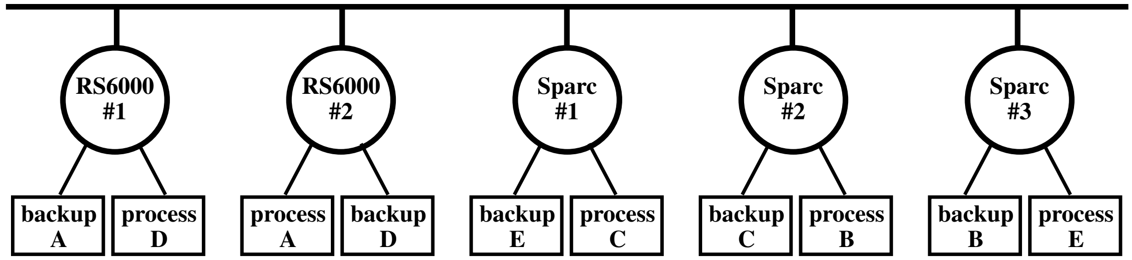


**Figure 3.A: Backup Creation**

*dpspawn() does not return until dpinit() creates the backup, using a modified internal form of dpspawn().*

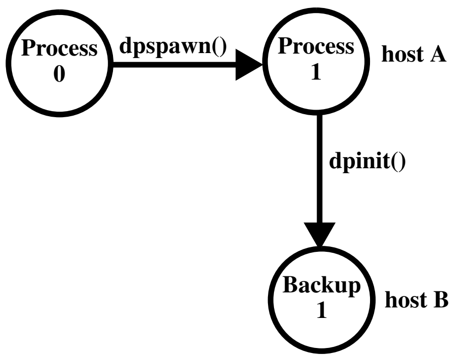



**Figure 3.B: Saving messages**

*Copies of reliable messages go to and are saved by the backup. Acks are sent by the backup, not the recipient.*

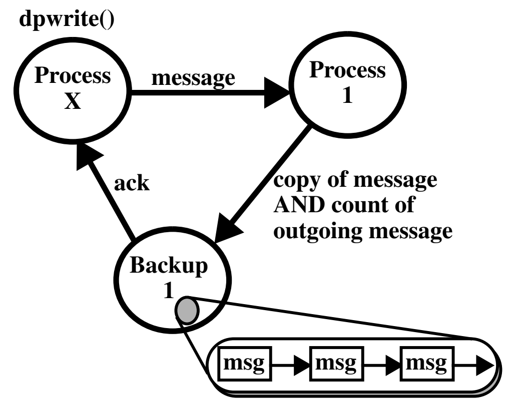



**Figure 3.C: Checkpoints**

*Periodically, a process saves its image in a file accessible to its backup, which is instructed to discard its saved incoming messages.*

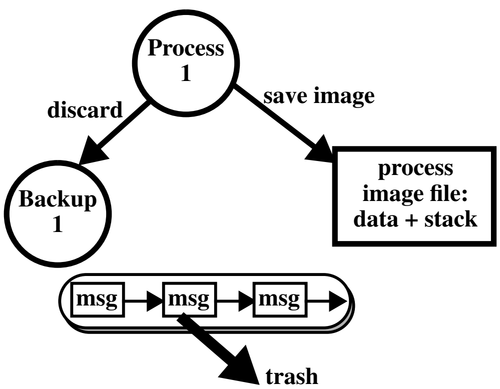



**Figure 3.D: Looking for Trouble**

*The backup process waits in dpinit(), sleeping and pinging the host on which the active process is executing. Timeou on ping means failure.*

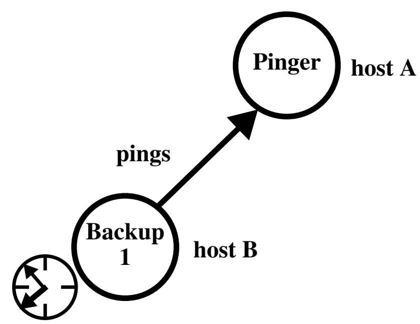



**Figure 3.E: Recovery— Phase I**

*Upon detecting failure, the backup process creates a new backup for itself.*

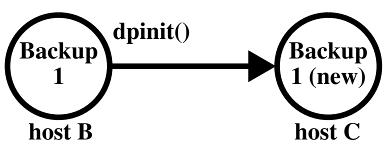



**Figure 3.F: Recovery— Phase II**

*The backup process then informs all other processes that it has taken over for the failed process.*

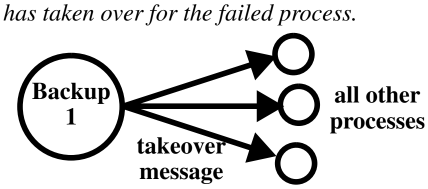



**Figure 3.G: Recovery— Phase III**

*The saved process image file is used to overwrite the backup's data and stack space— the backup now is executing as the original.*

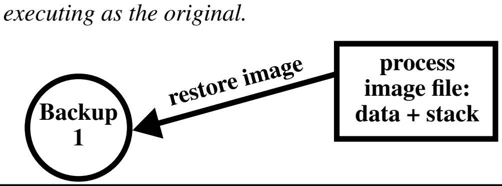
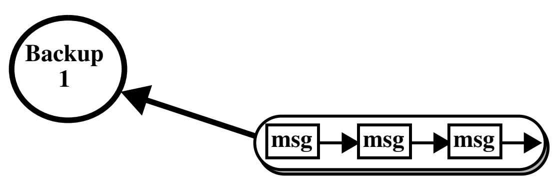



**Figure 3.H: Recovery— Phase IV**

*A period of "roll forward" ensures: the backup reads and processes the save messages.*



To preserve the total order of messages as B received them, B assigns each newly received message an internal sequence number. This number is passed on with every copy of that message that B sends to B'.

Failure is detected in an ad hoc fashion. If process B periodically sends dummy messages to its backup, B'. If B' does not hear from B after a time, it pings B's machine. If there is no respond after a number of tries, B' takes over from B.

Recovery begins with B' sending a *recovery* message to B's parent and every process that B has interacted with. Processes receiving such messages revise their internal DP process tables appropriately and propagate these messages to all other processes they in turn have communicated with. The application code layer never learns about this and will continue to use the original DPID of B which will now be mapped to that of B' by the DP implementation. Delay in this propagation of recovery messages poses no problem because messages sent to B will remain unacknowledged and hence be retransmitted, eventually, to B'. Furthermore, even if B has only failed temporarily (the transceiver cable fell out, say) and comes back to life in the middle of the recovery there still will not be a problem since it always forwards any messages that it receives to B'. Any messages that a temporarily reincarnated B would send will either be stale or cause the equivalent B' message to be treated as such.

Following the transmission of recovery messages by B', is the roll-forward phase. To avoid the delay that would result from having to roll-forward from scratch, active processes periodically (and transparent to the application) have checkpoints, where DP writes out the process's entire stack and data segment to disk. (This is why the active/backup pair must have some file space in common.) At the outset of the roll-forward phase of recovery, these segments are copied into the address space of process B', so the roll-forward starts from the last checkpoint. These checkpoints are done quite infrequently, on the order of 10-15 minutes. The rationale is that anyone running a computation on a group of workstations in which one fails should be grateful to have *only* a recover delay of 15 minutes. The reason for the fault-tolerance is for computations that run hours, not minutes.

To reduce excess message traffic resulting from redundant messages sent by B' in its roll-forward phase, the active process keeps B' informed as to the number of messages it has sent out to each process. During roll-forward, these counts are decremented and no messages are actually sent out to a given process until its corresponding count has reached 0.

This scheme necessarily requires a number of restrictions on the activities of the active processes. One severe restriction is that they cannot be doing I/O other than DP message transmission (or if they do I/O, its integrity can't be guaranteed).

## 6. Performance

A number of comparisons of DP's performance with that of PVM have been made. One test involves a ring of processes passing a single message from one to the other. Another involves a set of processes, each of which is sending and receiving messages symmetrically to all the others.

In each test, a DP and an equivalent PVM program were run simultaneously, in the same environment (same machines, same network, same directories, etc.) Each program was given a timeout value and the number of messages passed at that point was measured. Table 1 shows the ratio of messages sent in the DP program to that of the PVM.

**TABLE 1. Comparing DP and PVM**

| Processes | DP/PVM: ring | DP/PVM: set |
|---|---|---|
| 8 | 1.33 | 0.82 |
| 16 | 1.34 | 0.88 |
| 64 | 1.23 | 0.81 |
| 100 | 1.36 | 0.90 |
| 112 | 1.37 | 1.78 |

These results suggest DP performs comparably to PVM, that DP may scale better (probably because it does *not* rely on TCP connections) and that a more through performance study is desirable.

## 7. Portability

DP is implemented on SunOS, Solaris, AIX and on DEC RISCstations. Earlier versions were implemented on the Alliant FX-8 and the KSR-1. A Windows-NT implementation is underway as is a port to NetBSD.

Apart from its reliability enhancement, DP is very undemanding of the underlying system. It requires the ability to spawn remote processes, send messages without blocking and have interrupt-driven input. The reliability enhancement described above has been implemented on SunOS only. Presumably it could be carried out in most Unix environments.

## 8. Retrospect and Prospects

There seems to be a gap between system designers and application programmers in the area of parallel distributed programming. In this project, I started out wearing an application programmer hat. I had a set of requirements. There was no library then that came close to meeting them and even now no other library meets all of them. At the outset, I had no plan, for example, to provided dynamic process creation. As soon as I wore the system designer hat for a while, that seemed to be a great weakness in the design. It seemed that the flexibility to start with only a few processes and then as new tasks are identified, create additional ones is crucial. Ideally, the programmer would design the process structure of the application to mirror the logical task structure of the problem. After taking the trouble to provide this facility, I was quite chagrined to find that most of the DP users simply assess the number of machines that they have available, choose a number of processes about twice that number, and let them run, using a “worker parallelism” paradigm, in which worker processes are given or pick up tasks as they become idle. The reason for this is understandable. Available hardware is the determining factor in the plans of these practitioners. As I turned to using DP myself, I found I was doing the same thing. Is dynamic process creation really worth the trouble?

On the other hand, the interrupting message facility and the unreliable messages *are* used extensively. The former in particular has been seen to simplify the parallelization of existing code, by eliminating the need for finding the places in the code to put receives. The interrupting message handler takes care of that. Message sends still need to be inserted into the existing code, but somehow it is easier to identify the points where there is a result to brag about (to other processes say) than to identify receives. In cases where the *reverse* is true, then sending requests for data can be placed at the appropriate points and the *sending* of results can be interrupt driven.

On the other hand, the interrupting message facility and the unreliable messages *are* used extensively. The former in particular has been seen to simplify the parallelization of existing code, by eliminating the need for finding the places in the code to put receives. The interrupting message handler takes care of that. Message sends still need to be inserted into the existing code, but somehow it is easier to identify the points where there is a result to brag about (to other processes say) than to identify receives. In cases where the *reverse* is true, then requests for data can be placed at the appropriate points (send request in an interrupting message, wait for response) and the actual *sending* of results can be driven by the arrival of these interrupting messages.

The implementation of single processor fault-tolerance invites an effort to undertake process migration and load balancing. Whether the admittedly heavy-handed fault-tolerant scheme used here is efficient enough for that remains to be seen.

## 9. Availability

DP runs on Sun SPARCstations, DEC RISCstations and on IBM RS/6000s with C and Fortran interfaces. It is, along with documentation and some utilities, available from the author.


## 10. Acknowledgments

While an undergraduate at Harvard, Haibin Jiu spent two of his summers assisting in this effort. Jim Basney, a student at Oberlin spent a “winter term” on this as well. I especially would like to acknowledge the work of Jerry Chen, who while working on his doctorate at CUNY implemented an early version of DP on the KSR-1 and with whom I have had many valuable conversations.


## 11. References


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

Arnow, D.M.: Correlated Random Walks in Distributed Monte Carlo Programs. *ICIAM 91*, Washington D.C. (July 1991).

Arnow, D.M.: StdDP— a layered approach to distributed programming libraries. *T.R. 94-11* Dept. of CIS, Brooklyn College (1994).

Arnow, D.M., McAloon, K.M. and Tretkoff, C.: Distributed programming and disjunctive programming. *Proceedings of the Sixth IASTED-ISMM Int. Conf. on Parallel And Distributed Computing And Systems* Washington D.C. (October 1994).

Arnow, D.M, McAloon, K.M., and Tretkoff, C.: Parallel integer goal programing. *To appear in the 23rd ACM Computer Science Conference, Nashville*. (1994).

Aurebach, J., Kennedy, M., Russell, J., and Yemeni, S.: Interprocess communication in Concert/C. *T.R. RC 17341*, IBM Watson Research Center, Yorktown Heights, (1992).

Bal, H. E., Steiner, J. G., and Tanenbaum, A.S: Programming languages for distributed computing systems. *Computing Surveys 21*,3 (Sept. 1989).

Borg, A., Baumbach, J., and Glazer S.: A message system supporting fault tolerance. *9th ACM Symp. on Operating Systems Principles.* Bretton Woods, New Hampshire, (Oct. 1983).

Butler, R., and Lusk, E.: User's guide to the P4 programming system. *Tech. Rep. ANL-92/17*, Argonne Nat. Lab. (1992).

Chen, J.: *Distributed Green's function Monte Carlo calculations*. Ph.D Thesis, Dept. of CS, CUNY (1994).

Clark, K.L.: PARLOG and its applications. *IEEE Transactions on Software Engineering SE-14*, 12 (Dec. 1988).

Douglas, Craig C., Mattson, Timothy G., and Schultz, Martin H.: Parallel programming systems for workstation clusters. *Yale University Dept of CS Technical Report*, (Aug., 1993).

Geist, G.A. and Sunderam, V.S.: PVM— Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience 4*(4) (Jun., 1992).

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

Goldberg, Arthur P.: *Concert/C Tutorial: An Introduction to a Language for Distributed C Programming*, IBM Watson Research Center, Yorktown Heights, (Mar., 1993).

Hoare, C.A.R.: Communicating sequential processes. *Communications of the ACM 21*,8 (Aug. 1978).

Marsland, T.A., Breitkreutz, T., and Sutphen, S.: A network multi-processor for experiments in parallelism. *Concurrency: Practice and Experience, 3*(1), (1991).

MPI Forum: *Message Passing Interface Standard (Draft)*. Oak Ridge National Laboratory. (Nov. 1993).

Parsons, I.: Evaluation of distributed communication systems. Proceedings of CASCON '93, Vol 2. Toronto, Ontario, Canada (Oct. 1993)

Strom, R.E. and Yemeni, S.: NIL: An integrated language and system for distributed programming. *SIGPLAN Notes 21*, 10 (Oct. 1983).

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

## 12. Author Information

**David Arnow** is an Associate Professor in the Department of Computer and Information Science at Brooklyn College. He received an A.B. in Chemistry from Oberlin College in 1973 and a Ph.D. in Computer Science from NYU in 1981. His interests, as one might gather, include parallelizing scientific and operations research programs in a distributed environment. He is also deeply concerned with teaching Computer Science to undergraduates, regardless of its cruelty. His address is *arnow@sci.brooklyn.cuny.edu*.

## 13. Appendix: The DP interface header:

```
/* SEND FLAGS*/
#define   DPRECV     0x00 /* not interrupting */
#define   DPGETMSG   0x01 /* interrupting     */
#define   DPREL      0x00 /* guaranteed delivery */
#define   DPUNREL    0x02 /* no guarantee     */

/* RECV MODES*/
#define   DPBLOCK    0x00 /* Wait for message  */
#define   DPNOBLOCK  0x01 /* Don't wait        */

#define   DPSUCCESS 0     /* RETURN CODES*/
#define   DPFAIL    (-1)
#define   DPNOMESSAGE(-2)

#define   DPIDSIZE  28
typedef struct {
     char dpid_info[DPIDSIZE];
     } DPID;

typedef void   (*FUNCPTR)();
#define NULLFUNC((FUNCPTR) 0)

int dpinit(char *prog, char *semanticp,
     int *size, *hostid);
     /* RETURNS: DPFAIL or # of available hosts     */

int dpaddhost(char *hstn, *dmnn, *path,
     *user, *passwd);
     /* RETURNS:  number of hosts in host table     */

void dpgethost(int hid, DPHOST *hptr);
     /* STORES BACK: host info for host #hid        */

int dpwrite(DPID *dest, char *data,
     int nbytes, mode);
/* RETURNS: DPSUCESS or DPFAIL or DPDESTDEAD*/

int dprecv(DPID *src, char *data,
     int limit, int flags);
/* RETURNS: DPSUCCESS or DPFAIL or DPNOMESSAGE*/

int dpgetmsg(DPID *src, char *data,
     int limit);
/* RETURNS: DPSUCCESS or DPFAIL or DPNOMESSAGE*/

int dpspawn(char *prog, DPID *newid,
          int hid, char *semantic, int size,
               int sendflag);
/* STORES BACK the id of  the new process and
     returns DPSUCCESS or DPFAIL                    */

FUNCPTR dpcatchmsg(FUNCPTR f);
/* RETURNS NULLFUNC or ptr to previous catch function*/

void dpexit(char *exitstrng);
     /* removes process from DP group and exits*/
void dpgetpid(DPID *myid);
     /* STORES BACK: DPID of executing process      */
void dpalarm(long t, FUNCPTR f);
     /* set alarm for user's function f             */
void dplongjmp(jmpbuf label, int rv);
     /* longjmp to label, return value rv           */
void dppause(long t, FUNCPTR f);
     /* set alarm for user's function f and pause*/
int dpblock(); /* disable interrupts         */
void dpunblock();/* enable interrupts        */
void dpsetexfun(FUNCPTR f)
     /* set a f to be called when exiting    */
void dpstop(char *stopmsg);
     /* abandon ship fast                    */
int dpidmatch(DPID *ip1, DPID *ip2);
     /* return true if match; else false     */
```