DCL: Protocols and Primitives for Distributed and Concurrent Computing in Networked Environments*

V. S. Sunderam

Department of Math & Computer Science
Emory University, Atlanta, GA 30322, USA
vss@mathcs.emory.edu

Abstract

We present design and implementation strategies for providing general purpose distributed computing primitives on computer networks. This suite of primitives is intended to be a framework within which distributed and concurrent applications may be built in networked environments, in the absence of a distributed operating system. The proposed constructs are derived from typical application requirements, and include group communications, synchronization and recovery, and integrated distributed primitives such as mutual exclusion and consensus. We define an extensible suite of general purpose distributed computing primitives, discuss algorithms for their implementation, and present performance results and experiences.

1. Introduction

Distributed applications that execute on networks of computer systems are rapidly increasing in number and variety. Applications that are inherently distributed are evolving, and traditional applications are changing to exploit the many benefits of distribution. Developing such applications however, is a relatively complex task, and this difficulty sometimes discourages the construction of distributed systems. In this work, we address some of these issues, and propose a suite of general purpose distributed computing primitives to assist in the systematic development of distributed and concurrent applications. A strongly pragmatic approach has been adopted, and we present algorithms for the practical realization of these primitives, and also describe our experiences with a test implementation.

Developers of distributed or concurrent applications have several options available to them. While the distinctions are often blurred, there are generally three options available:

Distributed Operating Systems such as Amoeba[1] and Chorus [2] consist of a specialized kernel that executes on all the hosts on a network, and collectively manages its resources. These systems mandate that the distributed kernel must execute on all the system nodes — a limiting factor for many applications and environments.

Distributed Programming Environments such as ISIS [3], Marionette [4], PVM [5], and Linda [6] consist of a software layer that executes above the individual operating systems of autonomous machines, and provides various distributed facilities. While these systems are more flexible than distributed operating systems, they too have disadvantages, e.g., limited applicability, unfamiliar computing paradigm, or high overheads.

Low Level Programming is, interestingly, a frequent choice of distributed applications software. This approach is often attractive, though cumbersome, and a significant number of applications are implemented using these relatively low level networking or remote execution functions.

This latter class of distributed applications would benefit substantially from a comprehensive suite of distributed processing constructs, using standard, well defined interfaces, that are capable of being implemented on general purpose operating system environments. In this paper, we describe such a suite of distributed primitives, as well as algorithms for their implementation, and report on initial performance measurements. While it may be argued that such constructs can be built when needed, factors such as efficiency, complexity of some distributed algorithms, and duplicated effort make it undesirable to develop these on a per-application basis.

2. A Distributed Computing Framework

The functions that are required in the construction of distributed applications may be broadly classified as: (a) those that deal with inter-process communication; (b) synchronization related mechanisms; and (c) functions pertaining to process management, naming, and status checking or monitoring. This project has attempted to formulate a concise but comprehensive set that may be realized in a straightforward manner in a variety of environments.

2.1. Design Principles

The design of the constructs described herein, termed DCL (distributed computing library), has been guided by the following principles and assumptions:

- Nature of Target Environments: Computing platforms consisting of a variety of heterogeneous, autonomous machines interconnected by high- speed networks are, and will continue to be the norm; they universally support bare data transport, machine and process (or port) addressing, and usually a few higher-level mechanisms such as RPC.
- Asynchronous, Multiway Computing Model: The DCL primitives are oriented towards a model of *asynchronous* communication among *multiple* participants, as opposed to the traditional request-response or client-server paradigm which typically caters to pairwise interactions.
- High Level Abstractions, Failure Resilience, and Access: DCL goals are to enable straightforward access via a procedure call interface, support communications rollback facilities, and provied a suite of integral global operations.

2.2. Related Work

Most of the research efforts that are related to the DCL project are integral to distributed programming systems and environments, and are therefore not directly comparable. Nevertheless, many systems do support subsets (sometimes

* This material is based upon work supported by the National Science Foundation, under Award No. CCR-9118787, and the U. S. Department of Energy, under Grant No. DE-FG05-91ER25105.

intersecting) of the facilities provided by DCL. One example is ISIS[3], a distributed environment centered around replication and consistency. ISIS primarily supports the client-server paradigm, and uses atomic broadcast as the primary communications mechanism. The Chare kernel [7] is aimed at providing a unified paradigm for both shared and distributed memory machines and is oriented towards parallel processing applications. The Noreco system [8] is a language based system that supports CSP-like constructs for message interchange. In contrast, the DCL suite is not restricted to the client-server model, but rather, is based on asynchronous group interactions. The provided interface is suitable for use with any procedural language, thus enabling straightforward use. High level distributed computing primitives, including novel functions such as polling and distributed fetch-and-add, are integrally supported in DCL. Finally, the semantics of many of the DCL primitives are designed to permit program level failure resilience.

2.3. DCL Architecture

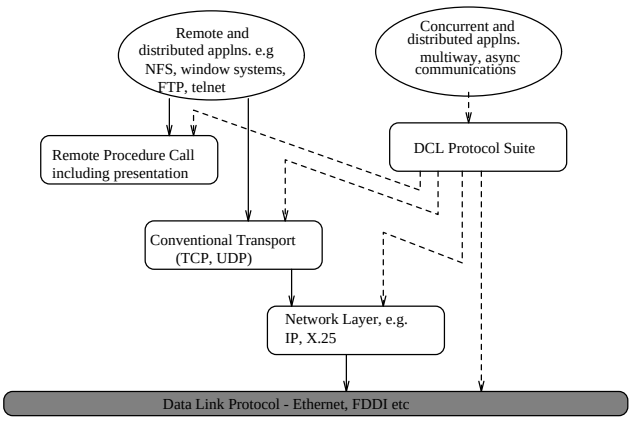

Figure 1: Architectural Overview of the DCL Protocol Suite

The DCL suite is intended to complement existing protocols for distributed and remote communication facilities, and provides services of an inherently different nature. As shown in Figure 1, the DCL primitives will be utilized by distributed and concurrent applications that require a multiparty communication model, asynchronous transfers, and high level abstractions for synchronization and other distributed functions. The DCL suite may be implemented using services provided by other existing protocols or a combination thereof; such an architecture permits the use of the most appropriate and efficient lower layer, while exploiting the infrastructural support of existing protocols when necessary.

3. The User Interface

In general, a distributed system may be viewed as a collection of communicating entities. Coordinating these entities effectively is the predominant responsibility of the supporting infrastructure. The primary objective of the DCL suite is therefore to provide a unified, standard, interface to a concise but comprehensive set of distributed functions.

3.1. Orthogonal Services

The DCL suite regards two important classes of distributed computing primitives as orthogonal services. The first is process initiation and termination; process semantics, remote execution, and related issues are overly dependent upon host operating systems to be generalized effectively in distributed computing primitives. The other aspect not directly addressed by DCL is naming. In the DCL suite, addressing is through the use of symbolic, string valued identifiers, augmented by integer valued version numbers for some constructs. Typically, many applications employ predefined, static symbolic names for addressing and may directly use the DCL constructs without the need for an intermediate name service mechanism.

3.2. Groups and Sessions

Most distributed systems may be viewed as being composed of *groups*, where a group is a collection of related communicating entities. The V-system provides a realization of the group concept [9], where a collection of processes form a group that may be viewed as a single logical entity. More recently, 'groupware systems', which support more general cooperation services, are receiving attention. Real-time conferencing and elections are examples of groupware applications; a representative description may be found in [10]. Recently, distributed computing systems have also been used for scientific problem-solving [11] and stochastic simulation [12]. In these applications, the notion of groups provides a convenient abstraction and structural framework for dealing with the components of the system. In the DCL suite, a more dynamic notion of a group is used. Processes that are components of a distributed application may *enroll* in *sessions*, in order to cooperate with other enrolled members. Each component is termed a session *participant*; a session is equivalent to a logical multiparty connection between enrolled participants.

3.3. Group Formation and Session Establishment

Processes that wish to cooperate using the DCL primitives are required to first enroll in a session, using the **`dcl_enroll`** construct. Session establishment is symmetric, and achieved through a quorum of enrolled participants. The arguments to the **`dcl_enroll`** construct include a string valued session name, limits on the number of participants in the session, and session parameters, e.g., synchronization window size, timeout values, default delivery semantics. The latter may specify idempotent, at most once, or best effort delivery.

As an example, consider a client-server application in which a single server receives and processes requests from a limited number of clients. The construct

```
dcl_enroll("dbtrans",     /* Session name */
       2, 16, 5,          /* 2-16 procs, 5 sec tmout */
       3,                 /* Rollback upto 3 msgs */
       ONCEONLY | SERVER, /* delivery, special */
       &sess_id, &my_id)  /* Locations unspecified */
```

might be executed by a server willing to participate with up to 15 clients simultaneously. Client processes would invoke the construct with the same arguments, except for specifying the `SERVER` flag and indicating `DONTCARE` as the maximum number of participants. The synchronization window argument (valued 3 in the example) is one of the DCL features that enables program level error recovery. Participants using this feature may request a rollback when required, resulting in a specified number of messages (subject to the limit) being redelivered to the invoker.

3.4. Message Transfer

Participants that have successfully enrolled in a session may exchange messages with any, some, or all other session members. The DCL suite supports strongly typed, message oriented data transfer (as opposed to byte stream or datagram delivery), and no restriction is imposed on message lengths. Several levels of reliability and delivery semantics are provided on a per-message basis, which may override the default options determined at session establishment time. In addition, the DCL suite supports a "quarantining" facility — each of a group of messages may be tagged, causing them to be buffered and delivered to the recipients as an aggregate.

```
dcl_sendmsg(<session id>,          /* From enroll */
        <list of participant ids>,  /* Or (A)BROADCAST */
        <delivery semantics flags>,/* Idempotent, once only etc */
        [<type>,<ptr>,<len>]        /* Data details */
        . . . . .                   /* May be repeated */
        <activity id & flags>)      /* For quarantining */
```

The destination argument in the **dcl_sendmsg** primitive may specify BROADCAST or ABROADCAST, both of which deliver the message to all enrolled participants. The latter specifies atomic broadcast semantics[13] — either all or none of the participants receive the message, and multiple atomic broadcasts are delivered to all recipients in the same order. This facility may only be used with "at most once" delivery semantics and may not be quarantined.

To receive messages, the **dcl_recvmsg** construct is used. Two arguments to this primitive are the session id, and a restricted list of message originators if desired. The receiving area may be specified as in the **dcl_sendmsg** construct, e.g.,

```
dcl_recvmsg(s_id,FROMANY,FLOAT,fptr, &sz)
```

or using a generic buffer for the incoming message as in

```
dcl_recvmsg(s_id,FROMANY,GENERIC,buf,&sz)
```

3.5. Synchronization

Participants in a DCL session may use any of several synchronization mechanisms during their operation. Some of these synchronization constructs require names; once again, these are string valued quantities, which are either known to the participants or obtained through an orthogonal service. Participants may obtain a lock on some abstract resource by invoking

dcl_mutex(<*s_id*>,<*resource*>,<*options*>);

The second form of synchronization provided in DCL is rendezvous based on events. Participants who invoke **dcl_waituntil** are suspended until the **dcl_event** construct, with the same argument, is executed by a session participant. Barrier synchronization, a feature frequently used by applications , is provided in DCL in both strict and quorum forms.

3.6. High Level and Miscellaneous Functions

While it is difficult to provide an all-inclusive collection of high level functions, those supported by DCL are the most frequently used, as determined by an informal analysis of a large collection of applications in different problem domains. The DCL construct **dcl_gather** is the simplest, and performs global aggregation, with variants that accomplish global extrema finding and global sum and product operations. In these primitives, each of a number of participants provides a vector containing local values; upon return, the vector is replaced with the results of the global operation. The DCL suite also provides a global operation that is best described as 'polling'. In this operation, one participant has a "question" that must be asked of all others, each of which must respond with an "answer". In the implementation of distributed applications, status checking to determine system state is often desired; achievable by the **dcl_status** construct. Although the DCL library is primarily intended for use in an asynchronous setting, it does contain facilities whereby the RPC or client-server paradigm may be used in a straightforward manner. The **dcl_export** primitive permits the declaration of a procedure that may be invoked by an external, remote participant; **dcl_invoke** forms the counterpart.

4. Implementation Aspects and Strategies

The DCL constructs provide a straightforward and flexible framework for distributed programming in networked environments. Implementation of the DCL primitives is, in several respects, similar to implementing high level network protocols. In software realizations of protocols, the usual options are to embed protocol mechanisms within the operating system kernel, or to implement them in a free running ''daemon'' process. Implementation purely in library form is generally not possible, since the required asynchrony cannot be provided. Exceptions do exist; when specialized operating system mechanisms such as upcalls [14] are available, protocols may be implemented as procedures or libraries. In this section we present (using a brief narrative description), some implementation algorithms for DCL in terms of an abstract *DCL provider entity*, which executes on each (participating) host on a network.

4.1. Implementation Model

We describe the implementation related aspects of the DCL suite in terms of a *DCL provider entity*, which executes on each host in a network. The implementation model uses the notion of a *host pool*, a collection of interconnected hosts on which the DCL provider is presumed to be operational. The DCL provider entity executes asynchronously and handles incoming requests from user processes as well as from the network. The provider entity contains two input queues, one for user process requests and the other to hold incoming messages from the network. It also contains one output queue to the network, and one output queue to each local user process. Each provider entity is assumed to be capable of sending messages over the network to any other provider entity on remote hosts.

4.2. Centralized vs. Distributed Control

In any distributed system, decisions concerning the extent to which control is vested in one, some, or all of the participants are very important. The strategy proposed in the DCL implementation model is a pragmatic one, derived from a combination of the centralized and fully distributed models. In essence, we attempt to provide both high efficiency and a high degree of fault tolerance, by using a centralized strategy when an operation is being performed. However, information regarding the state of that operation is distributed to all DCL providers soon thereafter — either using low priority messages, or by piggybacking this information on data transfers or synchronization messages.

4.3. Broadcast Among DCL Providers

In order to provide the DCL services in an efficient manner, a logical interconnection structure is imposed on the DCL provider entities. Each DCL provider is numbered with consecutive integers starting at 0, and they are assumed to form a logical broadcast spanning tree, with provider 0 at the root. In a system with N providers (hosts), each provider **`DCL[i]`** has a bi-directional link to providers **`DCL[j]`**, where $j = i \oplus 2^r$, for all $0 \le r < \lceil logN \rceil$.

4.4. Session Establishment

The basic steps that are to be performed by the DCL providers in order to establish sessions are (a) ensuring that a quorum of user entities issue **`dcl_create`** requests, and (b) the assignment of a session id. Session creation may be requested either with an explicit indication of participant locations or in "wildcard" terms; the DCL providers execute different algorithms in each case.

4.5. Data Transfer

Many of the implementation aspects dealing with the provision of data transfer facilities in DCL are well understood, including mechanisms for fragmentation and re-assembly, window based flow control, and sequencing for duplicate and loss detection. The latter issue may be part of a DCL implementation, depending on the layer above which the DCL suite is implemented. Data transfer mechanisms specific to DCL include typed data conversion, quarantining strategies, and synchronization and rollback. These may be implemented in a straightforward fashion, but have proven to be more efficient and convenient to manage when integrated with the core data transfer functions.

4.6. Error Detection and Recovery

The DCL suite provides for error handling in two main respects. First, failure of DCL provider entities are detected, and reconfiguration of the host pool is performed by the active provider entities. Invocation of DCL primitives subsequent to provider failures will return indications regarding the status of application sessions, data transfers, and synchronizations that may have been in progress. The second area where DCL provides failure handling capabilities is in the context of some of the DCL primitives. For example, in the case of quarantined data transfers, a receiving DCL provider entity may request a rollback, which causes re-delivery of previous messages as dictated by the synchronization window size. Within synchronization primitives such as mutual exclusion or barriers, DCL provider entities detect potential deadlock situations and provide suitable indications to application processes.

5. Preliminary Results

This section reports on preliminary results obtained from an experimental implementation of the DCL suite. The experiences reported pertain to the performance and behavior of DCL service functions, and are contrasted to identical functions implemented using an existing protocol, namely TCP. DCL was implemented on a 10Mb/s Ethernet network of SparcStation workstations. A protocol provider process executed on each workstation in *user* space; user entities communicated with the provider entity using process-to-process shared memory. The DCL provider process consisted of receiver and sender components; (simulated) asynchrony in their operation was achieved by the use of threads supported by the SunOS lightweight process library.

5.1. Functional Aspects

The DCL suite was used in experiments to implement several distributed applications, and compared against implementations that were based on existing facilities. One of the applications was a distributed version of a branch-and-bound algorithm for solving the 15-puzzle. This application was programmed using a master-slave paradigm, with the master process sending work to slave processes that explored a certain portion of the state-space tree and returned results. This particular application was straightforward to program using both DCL and the native RPC facility — similar levels of effort were required, and the implementations performed at approximately the same levels.

The branch-and-bound application possesses a simple communication structure. By contrast, the second application, a distributed matrix factorization algorithm, depends heavily on multicast and broadcast communication, and frequently performs barrier synchronization. This application was far easier to implement using DCL than any of the existing facilities, including RPC and stream protocols. Using the latter, managing the communications and synchronization was extremely tedious and difficult to implement, and the application performed slower than a DCL implementation by a factor of two. The consensus and broadcast primitives supported by DCL were of particular value for this application.

5.2. Performance

The performance of the DCL implementation was measured for primitive functions, as well as at the application level. As can be expected, the performance of the DCL protocol was superior to that of TCP for those functions involving one-to-many or many-to-many communications, since these have to be built from pairwise facilities in the stream protocol. A case in point is session establishment, timings under DCL ranged from 12 to 300 milliseconds (for 2 to 32 participants), while TCP required times ranging between 22 and 510 milliseconds.

No. of participants	Message Size (bytes)									
	100		512		1024		2048		4096	
2	9	(18)	12	(25)	14	(33)	22	(47)	33	(56)
4	13	(37)	21	(42)	21	(46)	34	(62)	41	(79)
8	25	(60)	30	(75)	30	(88)	46	(107)	61	(140)
16	38	(98)	45	(124)	48	(130)	71	(185)	95	(199)
32	69	(181)	75	(213)	76	(249)	101	(295)	128	(342)

Table 1: Data Transfer Times in DCL

Table 1 shows measured data transfer times using DCL, with corresponding TCP times shown in parenthesis. From the data transfer times shown, it can be seen that DCL outperforms TCP for small, message oriented data transfers, usually by a factor of 2 or 3. This disparity is probably due to the pairwise nature of the TCP transport protocol, which is inherently unsuited for multiway communication. Data exchanges with multiple simultaneous transmitters also showed similar ratios, indicating that TCP (indirectly) suffered as much as DCL did from Ethernet collisions and lost packets.

The implementation of high level distributed primitives in DCL also proved to be reasonably efficient. As can be anticipated, operations involving many-to-many communications, such as consensus and barrier synchronization, are the most expensive. Table 2 shows the times for consensus; it is likely that the sudden increase in timings for large numbers of participants is due mainly to network bandwidth bottlenecks and collisions.

No. of participants	Consensus data size(bytes) 0	100	512	1024
2	9	11	13	19
4	31	37	40	45
8	108	121	136	152
16	420	430	452	473
32	1500	1560	1640	1710

Table 2: Times for the consensus operation (ms)

The polling operation was, interestingly, more efficient than anticipated; timings are shown in Table 3. This leads us to believe that other distributed functions such as barrier synchronization would be more efficient if implemented using the polling operation, and experiments are in progress to determine this. In effect, the nature of the polling operation reduces the number of messages significantly (although with increases in the size of each message), thereby lowering the number of primitive data transfer operations that are required.

No. of parti-cipants	Message Size (Q&A; bytes) 8	64	128	512	1024
2	7	9	10	13	15
4	16	18	22	31	46
8	27	30	32	60	84
16	34	37	41	96	139
32	40	52	81	155	213

Table 3: Times in milliseconds for polling

6. Discussion

The DCL project is aimed at designing a protocol targeted towards the requirements of distributed applications. It is motivated by the belief that as distributed systems become more commonplace, a framework for their implementation in networked environments would be of significant value. DCL provides integral support for multiway communications, commonly required high level distributed computing primitives, and supports some degree of failure resilience. The DCL architecture allows for optimization using alternative layering strategies and efficient algorithms, and our preliminary results have shown that good performance is achievable even if the suite is implemented in user space. Particularly with imminent high speed networks, there will be a proliferation of distributed and concurrent applications that execute on general purpose networks, and DCL is likely to provide a valuable infrastructure for their deployment.

References

[1] S. J. Mullender, The Amoeba Distributed Operating System, *CWI Newsletter, No. 11:21-33, June 1986; No. 12:15-23, September 1986.*

[2] M. Rozier, et. al., The Chorus Distributed Operating System, *Computing Systems*, Vol. 1, No. 4, pp. 305-370, December 1988.

[3] K. Birman and K. Marzullo, ISIS and the META project, *Sun Technology*, 90-104, Summer 1989.

[4] M. Sullivan and D. Anderson, Marionette: A System for Parallel Distributed Programming Using a Master/Slave Model, *Proc. 9th Intl. Conf. Dist. Comp. Systems, 181-188, June 1989.*

[5] V. S. Sunderam, PVM : A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience*, Vol. 2, No. 4, pp. 315-339, December 1990.

[6] Gelernter, D., *et al*, "Parallel Programming in Linda", *Proc. Intl. Conf. on Parallel Processing*, IEEE, pp. 255-263, 1985.

[7] L. Kale, The Chare Kernel, *University of Illinois Technical Report*, June 1990.

[8] G. Spezzano, et. al., A Concurrent Programming Support for Distributed Systems, *Computing Systems*, Vol. 3, No. 3, Summer 1990.

[9] D. R. Cheriton, The V Distributed System, *Communications of the ACM*, 31(3):314-333, March 1988.

[10] C. A. Ellis, S. J. Gibbs, Concurrency Control in Groupware Systems, *Proc. 1989 SIGMOD Intl. Conf. on the Management of Data*, 399-407, June 1989.

[11] Fox, G., "Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech," *Concurrency: Practice and Experience*, Vol. 1, No. 1, pp. 63-103, 1989.

[12] V. S. Sunderam and V. Rego, ECLIPSE: A System for High Performance Concurrent Simulation, *Software: Practice and Experience*, Vol. 21, No. 11, pp. 1189-1219, November 1991.

[13] P. M. Melliar-Smith, *et. al.*, Broadcast Protocols for Distributed Systems, *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17-25, January 1990.

[14] D. Clark, An Alternative Protocol Implementation, *Systems Research Group Note*, Massachusetts Institute of Technology, May 19 82.

[15] S. Bogartz, Protocol Support for Distributed Groupware Systems, *Emory University Technical Report*, May 1991.

[16] V. S. Sunderam, The Polling Primitive for Hypercube Networks", *Emory University Technical Report*, March 1991.