Distributed Object-Oriented Programming With C++

Harold Carr, Robert Kessler, Mark Swanson
University of Utah
Department of Computer Science
3190 MEB
Salt Lake City, Utah 84112
801-581-6678
carr@cs.utah.edu

September 16, 1993

Abstract

Distributed C++ (DC++) is a language for writing parallel applications on loosely coupled distributed systems in C++. Its key idea is to extend the C++ class into 3 categories: vanilla C++ classes, classes which act as gateways between abstract processors, and classes whose instances may be passed by value between abstract processors via gateways. The 3 class categories are all written like vanilla C++ classes—declarations to the DC++ compiler augment gateways and "value" classes with operations which support their usage. An important result of making the 3 categories identical at the language level is that member function invocation of gateways looks identical to vanilla member function invocation. Therefore, the gateways may be redistributed (dynamically or statically) without program modification. Concurrency is achieved by creating multiple abstract processors and starting multiple threads of control operating within these abstract processors. DC++ transparently supports multitasking so the number of actual processors may change without program modification. This paper focuses on the DC++ language design with examples and performance measurements.

1 Introduction

Loosely coupled distributed systems built by interconnecting multiple workstations through a local area network (LAN) are an important resource increasingly available to programmers. LAN communication is becoming faster and such systems can easily be expanded to larger number of processors. DC++ is designed to exploit this resource for application programming.

Many researches have noted the compatibility between object-oriented programming (OOPS) and distributed programming [YT87, Yon90, Weg90]. Rather than design a new distributed object-oriented language ([BKT92]), we have chosen to create a distributed version of the de facto industry standard OOPS, C++ [ES90] (this paper assumes the reader is familiar with C++ syntax and semantics). DC++ provides a small number of simple extensions to C++, and have restricts some C++ idioms (such as global variables and traversal pointers). The restricted idioms are still available, but their results are undefined with respect to parallelism. The extensions are straightforward: DC++ provides 2 built-in classes: `DcDomain` and `DcThread`; and 2 new categories of class instance usage: *gateways* between domains, and "value" instances which may be passed between domains through gateway member function invocation and return.

DC++ supports parallelism by providing 2 types: *domains* and *threads* ([KCSS92, Swa92]). A domain is similar to a monitor [Hoa74]: only 1 thread of control may be active in it at any particular time. A domain is created by giving its constructor a physical processor number on which to allocate the domain. If the processor number is greater than the actual number of processors, the processor number chosen is the processor number modulo the number of processors in the system. This means that more than one domain may be explicitly (by the programmer) or implicitly (by the domain allocator) created on a processor. Therefore, domains may be thought of as *abstract processors*. The DC++ implementation supports multitasking so the programmer need not be concerned with these details. This gives the programmer the ability to initially develop their programs on a single processor, then to incrementally increase the number of real processors without program modification. A new thread is created to execute a function (or member function) in a specific domain. Once started, this thread may be chained or passed through multiple domains. The only restriction is that only one thread may be active in a given domain at any particular time.

DC++ *threads* communicate via *gateways*. A gateway object is created

within one domain. References to the gateway may be passed to other domains. Any gateway method invocations performed via these references will result in a remote procedure call (RPC) to the domain containing the actual gateway object. Since RPCs look identical to vanilla C++ member function invocations, redistribution of gateways is possible without program modification (modulo synchronization characteristics of the algorithm).

DC++ does not support global variables. They are problematical and, further, not necessary. A typical DC++ program has a "main" function which creates domains and gateways to those domains. It then creates threads in one or more of the domains by invoking a gateway member function. All access to member data is through member functions, as in vanilla C++. If it is necessary to have global variables, a programmer may create a gateway to a "global" domain, whose reference may be passed to other domains. Of course, the "global" variables may be partitioned into more than one gateway as determined by software engineering needs.

This paper gives an overview of the the object-oriented portion of DC++. We will use the dining philosophers problem [Hoa85] as a running example throughout this paper. We include performance measurements for this example. This paper does not discuss non-object-oriented features (such as explicit ports) or the implementation of DC++.

2 Domains — Abstract Processors

A domain is a logically encapsulated address and control object. It is created by specifying a physical processor number on which to allocate the domain.

```
DcDomain* domain[num_domains_needed];
for (i = 0; i < num_domains_needed; ++i)
        domain[i] = new DcDomain(i);
```

Since there may be more domains than actual physical processors, the domain is allocated on `i % DcNumPEs()`, where `DcNumPEs()` returns the number of physical processors available to the specific execution (specified as a command line argument to the DC++ applications program).

3 Gateways — Domain Entry Points

A gateway is a system wide unique reference (not a C++ reference) to an object created in a specific domain. It is treated as an ordinary C++

object: member function invocations on gateway objects result in RPCs if that gateway lives in a different domain than the domain from which it is invoked. Otherwise a vanilla C++ member function invocation results. A gateway class is declared like a vanilla C++ class, except it inherits from `gateway` (multiple inheritance, and virtual base classes are allowed but not discussed here).

```
class Footperson : public DcGateway {
        int num_phil_seated;
 public:
        Footperson();
        void SitDown(int phil);
        void GetUp(int phil);
};

class Fork : public DcGateway {
        int self;               // Name
 public:
        Fork(int i);
        void PickUpFork(int phil);
        void PutDownFork(int phil);
};

class Philosopher : public DcGateway {
        int         self;       // Name
        Fork        left;       // Fork at left hand
        Fork        right;      // Fork at right hand
        Footperson  footperson; // Table controller
 public:
        Philosopher(int i, Fork l, Fork r, Footperson p);
        void Live();
};

Philosopher::Philosopher(int i, Fork l, Fork r, Footperson p)
        : left(l), right(r), footperson(p)
{
        self = i;
}
```

A gateway instance is created on a specific domain by providing an optional (if not present the current domain is used) domain argument.

```
const int    num_phil = 5;
Fork*        fork[num_phil];
Philosopher* philosopher[num_phil];
Footperson*  footperson = new Footperson(new DcDomain(0));
for (i = 0; i < num_phil; ++i)
        fork[i] = new Fork(*new DcDomain(i + 1), i);
for (i = 0; i < num_phil; ++i)
        philosopher[i]
            = new Philosopher(*new DcDomain(i + 1 + num_phil),
                              i,
                              *fork[i]
                              *fork[(i + 1) % num_phil],
                              *footperson);
```

4 Threads

Concurrency is achieved by creating multiple threads of control.

```
for (i = 0; i < num_phil; ++i)
        new DcThread(*philosopher[i], Philosopher::Live);
```

This creates `num_phil` threads, each beginning execution in the domain associated with the `Philosopher` object, by invoking the `Live` member function on each object. If the member function accepts arguments they are given after the name of the member function.

5 Mutual Exclusion Synchronization

We ensure mutual exclusion synchronization by enforcing the rule that only one thread of control be active in a domain at a time. If another thread attempts entry to an occupied domain that thread will be queued on a FIFO queue for later entry when the domain becomes vacant. Note that we have unbundled domains (which ensure mutual exclusion), gateways (which provide entry points into domains), and vanilla C++ objects. This gives the programmer more flexibility. There may be a one-to-one mapping between gateways and domains if a program needs to lock on a per-object basis, or

there may be multiple gateways into the same domain. Further, domains may contain many instances of vanilla C++ objects and may pass and return DC++ value objects between domains. This gives the programmers choice while retaining the semantics of C++ objects.

6 Condition Synchronization

Threads synchronize implicitly through invocations of gateway member functions. Condition synchronization gives the programmer explicit control over these domain entry points. The programmer can specify that calling threads must block until a certain condition comes true before allowing entry. This is accomplished by associating *delay queues* with member functions.

```
Footperson::Footperson(){
        int num_phil_seated = 0;
        DcMakeDelayQueue(Footperson::SitDown);
}

Fork::Fork(int i){
        self = i;
        DcMakeDelayQueue(Fork::PickUpFork);
}
```

`DcMakeDelayQueue` associates an (initially open FIFO) delay queue for the given member function of the implicit (i.e., `*this`) or explicitly object given as an argument. Once created the programmer may control access to the member function by opening or closing the delay queue.

```
void Footperson::SitDown(int phil){
        if (++num_phil_seated == num_phil - 1)
                DcDQClose(Footperson::SitDown);
}

void Footperson::GetUp(int phil){
        if (--num_phil_seated < num_phil - 1)
                DcDQOpen(Footperson::SitDown);
}

void Fork::PickUpFork(int phil){
        DcDQClose(Fork::PickUpFork);
```

```
}

void Fork::PutDownFork(int phil){
        DcDQOpen(Fork::PickUpFork);
}
```

7 Thread Synchronization

Many times it is necessary for thread to created additional threads and then wait for the created threads to terminate (and perhaps return a value) before continuing. DC++ provides *placeholders* to meet this need. For example, to be able to obtain performance measures of the example program we need to start timing at the beginning of program execution and end timing when all threads terminate. This means that the initial thread which spawns the other threads must wait for them to terminate.

```
        DcThread* threads[num_phil];
        for (i = 0; i < num_phil; ++i)
                threads = new DcThread(*philosopher[i],
                                       Philosopher::Live,
                                       DcForValue);
        for (i = 0; i < num_phil; ++i)
                thread[i]->value();
```

Here the `DcThread` constructor is given an additional `DcForValue` argument. We capture a reference to the thread and then attempt to access the thread's *return* value. If the thread has not returned a value (implying termination) then the calling thread blocks until that value becomes available.

In our example, we have created 11 domains, 1 for each philosopher, 1 for each fork, and 1 for the footperson controlling entrance to the table of food. The philosophers obtain entry to the table and access of the forks by invoking the footperson and fork (gateway) member functions.

```
const num_iters = 50;

int Philosopher::Live(){
        while (num_iters--) {
                think(self);          // The work - spin
                footperson->SitDown(self);
                if (random()&01) {
```

```
                        left->PickUpFork(self);
                        right->PickUpFork(self);
                        eat(self);    // The work - spin
                        left->PutDownFork(self);
                        right->PutDownFork(self);
                } else {
                        right->PickUpFork(self);
                        left->PickUpFork(self);
                        eat(self);    // The work - spin
                        right->PutDownFork(self);
                        left->PutDownFork(self);
                }
                footperson->GetUp(self);
        }
        return 1;
}
```

When the philosophers terminate they return a value so the invoking thread can detect the termination of the spawned threads. A value need only be returned in the cases where termination needs to be detected or the thread computes a value to be used by the thread which spawned it.

8 Value Objects

Systems such as [B+91] provide primitives for sending and receiving data between processes, but require the programmer to pack and unpack aggregate data. DC++ provides a simple way for programs to pass (deep) copies of class instances between domains.

Our running example does not pass or return *value* objects. It does pass system wide unique references to domains and gateways, but these are handled specially by the DC++ compiler and do not fall into the category of value object. A value object is a class which inherits from the built-in DC class type `DcValue`. This tells the compiler that when one of these objects is passed by value as an argument to, or return value from, a gateway member function (passing pointers or C++ references via gateways is not allowed) it is to be totally copied. This means that any objects, pointers to objects, or built-in data types contained within the passed object must be recursively copied *and* any structure sharing present via pointers must be preserved. Any member data objects or member data pointers to objects

must be objects which also derived from `DcValue` so the compiler can add the necessary support for the complete copy operation.

```
class Test : public DcValue { // A base class.
        int I;
        ...
};

class Member : public DcValue { // A contained class.
        char C;
        ...
};

class Ing : public Test { // A derived class.
        Member  mi;
        Member* mp1;
        Member* mp2;
        double  D;
 public:
        Ing(int i, double d, char c)
                : Test(i), mi(c1), D(d), mp1(new Mem(c)), mp2(mp1) { }
        ~Ing() { delete mp1; }
};
```

Value objects may be passed-by-value freely between domains via gateway member function arguments and return values.

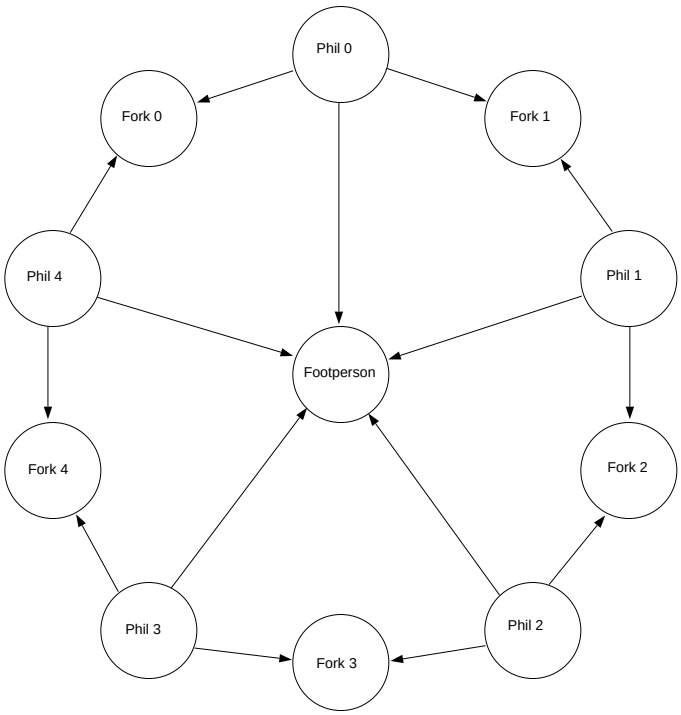
```
class Foo : public DcGateway {
 public:
        Entry(Ing I);
};

Ing Foo::Entry(Ing I) {
        I->some_member_function(...);
        return I;
}

Ing ing1(96, 4.5, 'i');
Foo* foo = new Foo();
Ing ing2 = foo->(ing1);
```

9 Performance Measurements

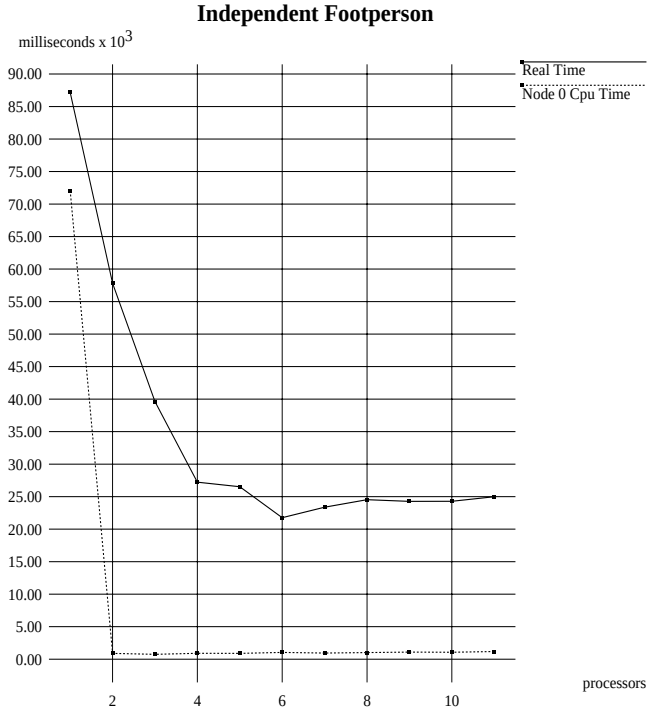
These measurements were taken by running the dining philosophers program on networked HP Series 9000, Model 370 workstations. The interdomain communication pattern is shown by the arcs from philosophers to forks or the footperson who controls access to the eating table.



Communications between processors is via BSD sockets. The thinking and eating done by the philosophers represents the actual work done by the program (in this case they are delay loops). For the purposes of these measurements, eating and thinking take the same amount of time for all processors.

We illustrate the speedup with 2 different allocation patterns. The base case for both is, of course, a single node containing all the forks, philosophers and the footperson. The best hand allocation (automated resource allocation is handled by [EK93]) is to always have the footperson domain reside on its own processor. Then, as the number of processors increases, put each philosopher domain on its own processor. When a processor must

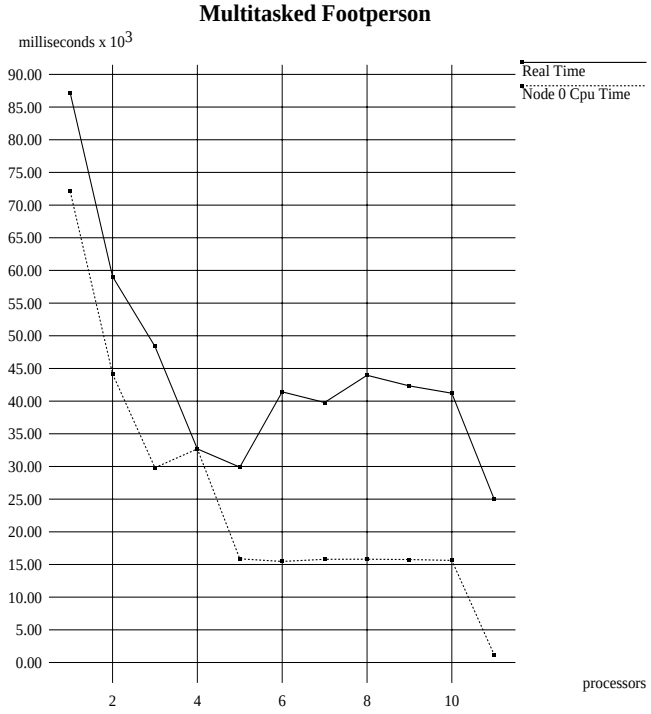
contain more than one philosopher domain, have 2 that share a common fork multitask on that processor. Also, put the shared fork domain on the same processor to reduce RPCs.



We can see that the CPU usage for processor 0 (which contains the footperson domain) drops to a steady state once the philosophers and forks are moved to other processors. We also see that the program running time drops as the philosophers are progressively allocated to their own processors. Optimal allocation is expected at 6 processors, where the footperson and each philosopher is operating on its own processor, with the forks multitasking on the philosopher processors. This is indeed the case, and when we allocate the forks to their own processors (for greater than 6 processors) we see that the program execution time increases, since access to forks then goes through (transparent) RPC.

The other allocation pattern is that shown in the example code in this paper. Here, we allocate the footperson to domain 0, the forks to domains 1-5, the philosophers to domains 6-11. For configurations where the number of processors is less than 11 the domains are allocated as stated earlier. In

this case the footperson processor is always multitasked (except when 11 processors are used) causing philosophers to needlessly wait when accessing the footperson, since another philosopher sharing the footperson's processor is multitasking with the footperson.



10 Conclusions, Status and Future Work

We have designed a distributed version of C++ based on concurrency mechanisms developed in our previous work in Concurrent Scheme [SK90]. We have a prototype DC++ compiler which transforms the language usage shown in this paper into C++, with calls to the DC++ runtime system. The result of the existing DC++ compiler must still be hand edited. We are continuing development of the compiler so it will fully automate the compilation process and detect illegal DC++ usages. The DC++ runtime system runs on homogeneous workstations. We plan to extend it to handle heterogeneous systems. The current performance measurements are very encouraging, especially since we still have many optimizations to make at

all levels of the system.

References

[B+91] A. Beguelin et al. A User's Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, 1991.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tannenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transaction of Software Engineering*, 18(3):190–205, March 1992.

[EK93] J. D. Evans and R. R. Kessler. Allocation of Parallel Programs With Time Variant Resource Requirements. In *Proceedings of the 1993 International Conference on Parallel Processing*. International Conference on Parallel Processing, The Pennsylvania State University Press, 1993.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[KCSS92] R. R. Kessler, H. Carr, L. Stoller, and M. Swanson. Implementing Concurrent Scheme for the Mayfly Distributed Parallel Processing System. *Lisp and Symbolic Computation Journal*, 5(1/2):73–93, May 1992.

[SK90] M. Swanson and R. Kessler. Concurrent scheme. In R. Halstead Jr. and T. Ito, editors, *Parallel Lisp: Languages and Systems, Lecture Notes in Computer Science Vol 441*, pages 200–234. Springer-Verlag, 1990.

[Swa92] M. Swanson. Concurrent Scheme Reference. *Lisp and Symbolic Computation Journal*, 5(1/2):95–104, May 1992.

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[Yon90] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

[YT87] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming.* MIT Press, 1987.