A C++/Linda Model for Distributed Objects

Rami Drucker and Ariel Frank
Department of Mathematics and Computer Science
Bar-Ilan University, Ramat-Gan, Israel, 52900
cradta@tau01.nsc.com, ariel@bimacs.cs.biu.ac.il

Abstract

Distributed systems have many advantages over centralized systems. We will focus on the novel Linda Tuple Space (TS) model designed for generative coordination in a distributed shared memory model. Currently, the Object-Oriented Programming (OOP) model is considered a most suitable model for handling complex computer-based systems. Among the OOP languages, the C++ language is a de-facto standard for developing advanced systems. This paper takes these two orthogonal computational models and combines them in a new way to form a C++/Linda model that supports transparent distributed objects. The idea is to create an integrated model that builds on the advantages of both models while complementing whatever is missing in each one alone. A prototype of the C++/Linda model was implemented and several distributed programs were successfully run on it.

1. Introduction

Distributed systems have many advantages over centralized systems. Among these advantages [2], one can point out the following: resource sharing, higher performance and throughput, availability and redundancy. Use of distributed applications is on the increase, beginning with networked applications, through financial and scientific computations, and recently groupware. Experience gained in designing and developing distributed systems shows that they are large and complex by nature, due to the required communication and coordination between the physically separated computers. Another realization is that adequate solutions require both the strength of parallel and distributed computing and the flexibility of the Object-Oriented Programming (OOP) model.

Currently, the OOP model is considered a most suitable model for handling complex computer-based systems. Better design, maintainability and reuse can be achieved by using OO features like data abstraction, encapsulation, inheritance and polymorphism. Among the OOP languages, the C++ language [13] is a de-facto standard for developing advanced systems. C++ does support concepts of software engineering but has no explicit linguistic constructs for parallel and distributed programming. However, local and remote objects do need to communicate between themselves. Consequently, it is difficult to construct concurrent C++ programs.

Existing Inter-Process Communication (IPC) and coordination mechanisms belong to two major models:

1) Message Passing model -- includes synchronous and asynchronous mechanisms, point to point and broadcast communication, and rendezvous and Remote Procedure Call (RPC) protocols.
2) Data Sharing model -- includes mechanisms of shared logical variables common in logic programming, and shared or distributed data structures available in programming languages supporting distributed shared memory.

Unfortunately, the model of message passing is lacking in regard to software engineering concepts. It requires the programmer to deal with explicit message formats, the locating of message destination addresses and the asynchronous nature of communication. Even the widely used RPC protocol has some disadvantages: the lack of an asynchronous procedure call and the possibility of RPC orphans [1]. On the other hand, the data sharing model is more difficult to implement because it requires data replication and consistency preserving mechanisms.

---

1 This research was done as part of a M.Sc. dissertation of the author.
We will focus on the novel Linda Tuple Space (TS) model [5] designed for generative coordination. Linda supports decoupled process creation and synchronization within a distributed shared memory model, in a simple and elegant manner. However, even the basic Linda model [1] has some disadvantages: the TS is unprotected since any process can access any tuple in it, and there is no hierarchical organization of tuples, therefore no sense of name scoping. This paper takes the following two orthogonal computational paradigms:

1) The OOP model as implemented by the C++ programming language,
2) The distributed shared data model as implemented by the Linda coordination language,
and combines them in a new way to form a C++/Linda model [4] that supports transparent distributed objects. The idea is to create an integrated model that builds on the advantages of both models while complementing whatever is missing in each one alone.

This paper is organized as follows. Section 2 presents this new C++/Linda model for distributed objects. Next, section 3 deals with the feasibility of the model and describes a prototype implementation. The experiments carried out with the prototype are presented in section 4. Section 5 reviews related work while section 6 concludes the article.

2. The C++/Linda model

Most mechanisms for object sharing do not support truly distributed objects. Rather, they offer the possibility to access a remote object either by means of object replication or by migrating an object between processes wishing to access it. There are however models that have attempted to combine Linda with OOP languages. In models such as C-Linda [5] and Eiffel-Linda [6], Linda primitives are invoked solely through explicit program calls. The C++/Linda model described here differs from these models by also supporting implicit Linda calls from within distributed objects.

The C++/Linda model proposed here (see Figure 1) combines C++ and Linda at two levels:

**Programmer Level** -- where explicit calls to Linda's primitives are carried out in the same manner as in C-Linda.

**System Level** -- where C++ objects implicitly call Linda’s primitives to access and manipulate their own object data.

<table>
<thead>
<tr>
<th>Linda</th>
<th>Programmer Level -- C++/Linda Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run</td>
<td>System</td>
</tr>
<tr>
<td></td>
<td>Distributed C++ Objects</td>
</tr>
</tbody>
</table>

**Figure 1. The C++/Linda model**

In the next subsection we describe three design alternatives for building the C++/Linda model. Each of these designs differ in the level of data and code distribution, the access transparency it provides and its implementation costs.

**2.1. The design alternatives**

The three alternatives for the design of the C++/Linda model are:

1) Implement C++/Linda in a manner similar to C-Linda while using C++ of course. The entire C++ program or its subprograms are replicated on system nodes. The program processes refer to shared data stored in the TS via explicit calls to Linda primitives. Naturally, the implementation costs of this option are relatively low but so are the gained benefits.

2) Most of the added value of the C++/Linda model stems from using the System Level that transparently supports distributed objects. The member data of distributed objects are stored in the TS while their member functions are replicated on each participating node (see Figure 2). The System Level implicitly stores each of the object's variables as a TS tuple with a unique key. Clearly, there is no need to use a Remote Method Call (RMC) since the methods of a distributed object are local to a participating node. We return to the design and implementation costs of this design alternative in sections 3 and 4.
3) Both member functions and member data of an object are distributed. This option "improves" on the previous option since the main process performs a Linda’s out(object) operation to the TS instead of replicating the object’s code on all participating nodes. This operation creates an instance of the object (both methods and data) in the TS, thus enabling other processes to perform in(object) or rd(object) operations to create a local copy of that object on their node when needed. The feasibility of implementing such a scheme has been demonstrated by [7]. Issues related to caching data and code on local nodes, such as cache coherence and on-the-fly code upgrades, are also relevant to this option but are further discussed in [4].

2.2. Description of the model levels

The C++/Linda model augments the C++ object-oriented programming language with the benefits of Linda’s TS. The idea is to take a C++ program and transparently replace its data storage mechanism with that of the TS. Distributed C++ variables are automatically stored as unique tuples in the TS rather than being allocated storage in internal memory (on a stack or in the heap). Figure 3 demonstrates a typical System Level sequence of operations executed for a statement that increments the distributed variable A.

![Figure 2. Layout of distributed objects](image)

![Figure 3. Statement execution in C++/Linda](image)

The C++/Linda model has many advantages and it provides solutions to many of the issues discussed earlier in this article:

1) Enabling concurrency -- sequential C++ programs can be enhanced to run in a distributed environment by using distributed objects.

2) Transparency -- processes wishing to share objects are not required anymore to transfer data structures between them. Additionally, there is no need for explicit message passing mechanisms, so the programmer is actually free to focus on the algorithmic part of the problem. Since processes refer to distributed objects using regular C++ syntax, we have both programming and access transparency. Additionally, calls to Linda primitives from
the Programmer Level are not required, but can be used if warranted.

3) TS access control -- processes do not read or write tuples directly. Rather, they manipulate distributed objects that contain private and protected data. Invoking the object methods in a controlled manner supports information hiding and prevents the "global variable" appearance of tuples in the TS.

4) Lack of name scoping -- use of C++ classes preserves name scoping rules by defining private and protected member data and functions.

3. Feasibility of the C++/Linda model

To demonstrate the feasibility of the C++/Linda model, a prototype was built by combining available C++ and Linda systems. The prototype was used in testing several distributed applications (see section 4). The decision as to which of the three design alternatives to use for the prototype was based upon evaluation of the efforts required for its implementation versus the benefits provided.

As can be seen from subsection 2.1, the first design option is too simplistic. Hence, the choice was narrowed down to options 2 and 3. It is clear that option 3 is the most comprehensive design solution for the C++/Linda model. However, its implementation requires use of relatively complicated mechanisms such as those used in DSR [7]. Therefore, the decision made was to implement an initial prototype based on option 2, mainly due to the reduced efforts and risks involved. Implementation options and details are described in the following subsections.

3.1. Finding C++ and Linda systems

To build the prototype, a C++ environment and a Linda system were required. We selected the G++ (GNU C++) compiler [12] because it is an excellent public-domain environment and its source code is available. Finding a Linda system was a harder task since commercial Linda systems are scarce and also unaffordable. Eventually, we settled on a public-domain Linda system called POSYBL [11], that implements Linda on a network of Sun workstations. POSYBL also comes with source code, allowing us to add necessary modifications to the system. At that point, the main issue that remained unresolved was how to best integrate C++ and Linda in building the prototype.

3.2. Tying C++ and Linda using a compiler

Most of the existing combinations between Linda and other programming languages are based on pre-compilers. However, here we have to intervene at the compiler level, not just at the pre-compiler level. Integrating Linda and C++ requires the addition of a new keyword to C++ to enable declarations and handling of distributed variables. For example, given the following code:

```cpp
volatile int a; // regular data type
distributed int b; // distributed data type
```

the compiler has to identify and treat the new keyword `distributed` like it handles the ANSI-C optional specifier `volatile`, for example. During code generation phase, the compiler has to act differently when it encounters distributed types. Instead of allocating storage for them in internal memory, it generates special code to create a new tuple for that variable and for inserting it into the TS. In addition, the compiler has to find all references to a distributed variable in the program and generate appropriate code to read and write the relevant tuples from and into the TS. Unfortunately, this implementation option requires changes to the lexical and semantical analyzers of the compiler, and to the code generator.

Consequently, it was decided not to use a compiler based implementation for this prototype. The same arguments hold true for a pre-compiler based solution. Additionally, a pre-compiler would have additional problems such as dealing with distributed variables inside expressions.

3.3. Linking C++ and Linda at run-time

The idea behind this alternative, is figuring out how and where the C++ run-time system accesses the member data of distributed objects. These accesses have to be modified to insert calls to special library functions. However, C++ programs usually get translated into C before entering the final code generation phase. Therefore, no type identification information exists at run-time, hence this solution is not feasible.

3.4. Using the C++ language to tie-in Linda

An interesting argument against the above two solutions for implementing the prototype follows. If C++ is indeed such a powerful and flexible programming language, then we should be able to
avoid the use of external means or the "bending" of the base language to support the Linda operations. The essential insight behind our implementation choice for the C++/Linda prototype adheres to this line of thought. We simply define a new class called distrib that contains all the necessary definitions and code to support the creation and handling of distributed variables. C++ is most suitable for implementing such a class since it allows redefinition of operations on distributed types via the operator overloading mechanism. To minimize coding and simplify support for all basic types, we also used the template class mechanism of the language. The solution chosen here proved to also be a very convenient approach for prototyping.

4. Experimenting with the prototype

To use the prototype, we start by taking a concurrent C/C++ program that runs correctly on a single processor with multitasking support. This concurrent program has to be transformed into a correct distributed program to be run on the C++/Linda prototype. The transformation process consists of the following steps:

1) Modifying the original program — a typical concurrent program consists of a main module responsible for system-wide initializations and for creating other processes. We begin by selecting the variables we wish to distribute. The declaration part of these variables is modified using the distrib<Type> keyword, where <Type> stands for the basic type of the original variable. Also, original code sections that use concurrency mechanisms need to be modified to use similar mechanisms provided by the prototype.

Example concurrency mechanisms currently implemented include distributed semaphore, distributed owner lock and distributed barrier [4].

2) Compiling the distributed program — the modified program is compiled just like any other C++ program, using the distrib class definitions.

3) Linking the distributed program — after compiling the program, its object files are linked with the POSYBL library that contains Linda primitives used by the (Programmer Level and) System Level of the C++/Linda model.

4) Running the program — the TS is started initially on every participating node. Then, the main task can be started. This main task creates all other tasks and distributes them over the network. The program is now run while using distributed objects.

To demonstrate the prototype capabilities, we will look at a distributed solution to the classic "dining philosophers" problem. We have here five philosophers that sit around a circular table. Each philosopher spends his life alternating between thinking and eating a portion from a large spaghetti plate. Unfortunately, the philosophers have only 5 forks but each of them needs two forks to eat a portion. We used the original solution based on representing forks as semaphores [1]. First, a concurrent C program using UNIX semaphores was written (see Listing 1). Then it was converted to a C++ program with the necessary modifications to make it distributed (see Listing 2). The results of the run were as expected of course.

5. Related work

Relevant related work comes naturally from the fields of distributed shared memory models, OOP languages and Linda embedded languages [4]. To date, researchers have been actively investigating the building of OO distributed systems. Emerald [3] and Electra [8] are two examples of such systems. However, these systems usually rely on RPC and RMC mechanisms or object migration. Several recent system designs have been based on the distributed shared memory model. For example, SO [9] uses predefined shared object types to coordinate between running processes. There has been research into integration of OOP languages such as C++ and Linda [10],[14] or the combination of Eiffel-Linda [6].

6. Conclusion

This paper describes the C++/Linda model that supports distributed objects in a C++ programming environment through use of the Linda Tuple Space. This necessitated dealing with the issues of combining OOP languages with distributed systems. To demonstrate the feasibility of the proposed model, a prototype was built by combining existing C++ and Linda systems in an elegant way. Several programs that were tested on the prototype proved the model feasibility.

This research can be extended by implementing a prototype that uses the third design alternative. A compiler based solution can also provide significant performance improvements. The C++/Linda model shows a feasible solution for supporting distributed C++ objects.
int  forks[5]; /* fork semaphore ID */
int  print_sem; /* serialize printout of tasks */

main(argc, argv)
  int  i;
  char  *argv[1];
{
  int  n, phil_num;
  char  buf[100];

  /* retrieve command line parameters */
  phil_num = atoi(argv[1]); /* get philosopher number */
  print_sem = atoi(argv[2]); /* get print semaphore ID */

  for (n = 3; n < argc; n++) /* get forks semaphore IDs */
    forks[n-3] = atoi(argv[n]);

  /* start with thinking state */
  P(print_sem);
  fprintf(stderr, "phil %i thinking", atoi(argv[1]));
  fprintf(stderr, "\n");
  fflush(stderr);
  V(print_sem);

  while (1) {
    sleep((rand())%8); /* sleep between 0 - 8 seconds */
    if (phil_num < 5) { /* philosophers 1 - 4 act the same */
        /* pick up right hand fork and then left hand fork */
        P(forks[phil_num-1]); /* right hand fork */
        P(forks[phil_num]); /* left hand fork */
    } else { /* philosopher no. 5 */
        P(forks[0]); /* left hand fork */
        P(forks[4]); /* right hand fork */
    }

  /* eating session */
  P(print_sem);
  fprintf(stderr, "phil %i eating", atoi(argv[1]));
  fprintf(stderr, "\n");
  fflush(stderr);
  V(print_sem);
  sleep((rand())%8); /* sleep between 0 - 8 seconds */

  /* thinking session */
  P(print_sem);
  fprintf(stderr, "phil %i thinking", atoi(argv[1]));
  fprintf(stderr, "\n");
  fflush(stderr);
  V(print_sem);
  if (phil_num < 5) { /* philosophers 1 - 4 act the same */
      /* put down right hand fork and then left hand fork */
      V(forks[phil_num-1]); /* right hand fork */
      V(forks[phil_num]); /* left hand fork */
    } else { /* philosopher no. 5 */
      V(forks[0]); /* left hand fork */
      V(forks[4]); /* right hand fork */
    }
  } /* end while */
} /* end program */

Listing 1: A Concurrent C++ Program
Listing 2: A Distributed C++ Program
References