Turk J Elec Engin, VOL.10, NO.1 2002, © TÜBİTAK

CPL: A Language for Real-Time Distributed Object Programming

Erhan SARIDOĞAN
Turkish Navy, Software Development Center
Arastirma Merkezi Komutanligi
81504, Pendik, Istanbul-TURKEY
e-mail: esaridogan@yahoo.com
Nadia ERDOĞAN
Computer Engineering Department
Electrical-Electronics Engineering Faculty
Istanbul Technical University
80686, Ayazaga, Istanbul-TURKEY
e-mail: erdogan@cs.itu.edu.tr

Abstract

As processing and time requirements of computer systems increase over the borders of single processor architectures, it is becoming more and more attractive to use distributed computing with additional real-time capabilities. In several cases, traditional programming languages have become insufficient to build distributed systems easily, especially when real-time issues and basic software quality factors are concerned. In this paper, a concurrent, object-oriented, distributed real-time programming language, CPL, with a supportive run-time system, namely the CORD-RTS, is introduced and new language features are described. The new language provides an efficient solution for command and control systems by embedding distribution and real-time issues within new language constructs. The language preprocessor translates these language constructs into portable C++ code to establish run-time connection with the RTS, which provides real-time communication between distributed objects.

Key Words: *concurrent, object-oriented, real-time, distributed, programming language*

1. Introduction

A real-time computation is one in which the timeliness of a response to an event is as important as the correctness of the response. Parallel and distributed computing is useful for real-time systems for three fundamentally different reasons [1]. First, the processing and response requirements of some applications are so extreme that even the fastest uni- processors are inadequate. It is natural to apply multiple processors/computers to such problems. Second, such systems may use multiple copies of system components, thus providing fault-tolerance through redundancy. Third, some real-time applications are by nature geographically distributed. In all cases, the real-time application requires techniques for partitioning the problem, coordinating concurrent access, and communicating within tight delay bounds. It is true that real-time distributed software can be developed in any language, but when conventional programming languages are considered, programming becomes difficult and time-consuming because developers have to implement timing constraints, communication and distribution issues through operating system primitives of the underlying computer system.

Real-time distributed computing systems started to evolve during the 1990s and today are one of the fastest-growing areas in technology with a strong demand for analysis, design and implementation of large-scale real-time applications. Recently, researchers have concentrated on the conceptual match between the object paradigm and real-time systems [2]. In contrast to procedural programming, which emphasizes algorithmic sequences, object-oriented programming uses a structure of collaborating parts or objects. Each part performs its specialized processing by reacting to inputs from its immediate neighbors. The superposition in time of these parts' localized behavior, then, results in overall system behavior. This approach fits very well with many real-time systems whose task is to transform external inputs into appropriate timely outputs. Many concurrent flows may pass through the system, but the internal structure of the system remains the same. This structure dominant style fits well with the object-oriented paradigm, which provides a framework through which behavior flows. The effective use of object-oriented methodologies to reduce the development complexity and maintenance costs of large-scale applications has also been a driving force in the integration of object-oriented design and real-time computing.

In practice, real-time systems support hard real-time activities, soft real-time activities, or a combination of both. Hard real-time activities require verifiable hard real-time techniques to decide whether a given system will meet all its deadlines. To guarantee timeliness, off- line scheduling algorithms under worst-case assumptions are used to determine a feasible schedule because determining if even a single deadline has been violated is crucial. Soft real- time activities, on the other hand, have moderate dependability requirements because missing a deadline has no fatal consequences. As a result, their timeliness aspects may usually be implemented by on-line techniques, such as on-line scheduling with static or dynamic priorities. Software designers of both classes of systems need tools and a methodology to develop reliable, efficient and robust real-time systems. CPL is the language we propose for real-time distributed object programming [3], [4]. We have selected the object-oriented methodology and enhanced it with programming concepts and techniques that support the software design and implementation of time/event driven real-time systems. CPL, standing for *CORD Programming Language*, gets its name from its underlying support environment, the *CORD-RTS* (Concurrent, Object-Oriented and Real-Time Distribution Run-Time System) [3].

As the target domain of our work includes systems with dynamically changing resources and loads, off-line analysis techniques cannot be applied to make definitive claims about deadlines; therefore CPL applies best to soft real-time systems. The static priority based online scheduling scheme of the underlying operating system, coupled with CORD-RTS prioritized message handling and time management mechanisms, insure the fulfillment of soft real-time requirements.

CPL uses the object-oriented programming approach. The essence of object-orientation is the unification of the type and module concept in the class construct. Thus, in CPL, a class is the basic unit of development and compilation. An object is the unit of concurrency. A CPL application consists of a collection of objects distributed over multiple nodes, interacting via remote method calls. Being a syntactic and semantic extension of C++, CPL provides a "general high-level programming style" that enables programmers to practice effective methods in distributed real-time programming. Distributed objects represent a higher-level structure for distributed applications. Communication between objects is transparent at the programming language level where there is no reference to the location of objects, thus providing the programmer with a single object space.

CPL enhances conventional object behavior with real-time extensions. CPL time- related constructs supervise timed events. They express the timing behavior of an application to guarantee the timeliness of a response to an event. Periodic method invocation, iteration statements with timing constraints, specification of timing requirements both at the statement level and on object communication, and an advanced exception control mechanism constitute a rich set of high-level and high-precision real-time attributes that CPL provides, which contribute to the development of reliable real-time distributed applications.

A cost-effective way to support object-oriented real-time distributed programs is to realize an execution environment by developing a middleware running on well-established commercial software/hardware platforms. CPL is supported by efficient middleware architecture, the CORD-RTS, which is implemented on a multi-node computer network executing Sun OS 4.1.3 and Solaris 2.5 operating systems. CORD-RTS provides all the functionality needed in realizing the behavior of CPL real-time objects.

CPL is supported by an extension of C++ that requires a new, extended compiler. Such a language extension requires serious work, but we believe it increases the productivity and quality of application programming. Thus, we have introduced new keywords that require a language translator for programmer convenience, rather than providing an API that wraps the services of the middleware.

2. Target Domain

A growing class of real-time systems includes applications on manufacturing process control, videoconferencing, command and control, large-scale distributed interactive simulation, real-time storage and search for information and real-time communication and display of information. Of these, Distributed Command and Control (DC^2) systems are in the main target domain of this study. Both industrial and military command and control systems require an efficient infrastructure capable of handling large amounts of high frequency data. Automated control systems collect data from various sensors or input devices, evaluate them and remotely control some actuators, preserving real-time constraints. In addition to efficient device control, fast and reliable access to shared data is also required. The design decisions related to the real-time features of CPL have been strongly guided by these requirements.

3. Related Work

Object-oriented programming is a widely used technique that divides programs into smaller parts, called objects, that can only be accessed via methods that are defined in their interfaces. In the context of distributed computing, objects are classified either as activities or data. Many concurrent languages, such as Mentat [5], RTC++ [6], Concurrent Smalltalk [7] and Eiffel [8], have chosen to use objects as processing elements subject to distribution. On the other hand, Linda [9] uses data objects while POOL-T [10] uses both types. Of these, Mentat and RTC++ are quite similar to CPL. However, Mentat does not have real-time features, while RTC++ does not have shared data and device access.

In a concurrent environment for coarse granular programming allowing distributable processing elements, objects are implemented as distinct processes that use the inter-process communication mechanisms of the underlying operating system. Concurrent languages, such as Mentat, ES-Kit C++ [11] and Pearl [12], have special constructs that provide communication in a synchronous or asynchronous manner. Ada-83/95, which is not a distributed language but a concurrent one, uses the rendezvous mechanism to exchange information between tasks existing within the context of a program. Distributed operating systems, such as Mach [13], hide the network level communication so that all programs seem to execute on a single machine.

Common Object Request Broker Architecture (CORBA) [14] is a widely accepted and rapidly developing standard for commercial products used in implementing distributed applications over heterogeneous computer networks by using object-technology.

Object-oriented real-time programming is a new area of interest for real-time software developers. Real-Time Java [15], Real-Time CORBA [16] and the TMO programming scheme [17] are examples of new distributed real-time tools that have been developed in recent years. Ada-95, having a more object-oriented approach than Ada-83, is one of the most powerful and widely used real-time, concurrent programming languages, especially for defense applications. However, separate Ada programs cannot communicate with each other, especially when distributed.

None of the existing languages provides support for concurrency, object-orientation, distribution, and real-time execution at the same time. In addition, CPL uses easy-to-learn keywords rather than libraries, enabling fast and reliable development for command and control systems.

4. The Cord System

An efficient middleware architecture named *CORD-RTS* (Concurrent, Object-Oriented and Real-Time Distribution Run-Time System) has been developed to support *CPL* (Cord Programming Language). CORD-RTS allows the development of distributed real-time applications that are independent of node hardware, operating system and network topology.

A CPL program consists of a main program with active, passive and device objects, all communicating with each other via the CORD-RTS, which provides network transparency and inter-object communication.

Each node runs a copy of RTS, which provides its functionalities through a set of manager processes, namely, the Object Manager, Net Manager, Device Manager and Error Manager. A command shell is provided to interpret user commands interactively for system control and monitoring. Several shell commands allow the user to load and start programs, to manually register classes and objects on-line, and to create or execute objects individually from existing classes. Users can also monitor and get information about the system and status of programs, classes and objects.

The structure of CORD-RTS and the interactions between a CPL program components and the CORD system are illustrated in Figure 1. The detailed description of CORD-RTS [18] is beyond the scope of this article, which focuses on CPL as a real-time programming language.

Distributed real-time computing necessitates the establishment of a global time base to supply a common time reference to all nodes executing a distributed application. As a reference time, the Global Positioning System (GPS) is widely used as it provides the highest level of accuracy with global coverage based on satellite-based radio navigation systems. However, it is not feasible to install a GPS receiver in each computer node for time reference. There are a number of solutions, both in hardware and software, to network level time synchronization. The current CORD System uses Network Time Protocol (NTP) Version 4.0 [19]. During tests, 200 to 800 microseconds of accuracy were achieved in synchronizing workstation clocks on an ATM network by using the internal clock of one of the workstations as the main time reference.

5. CPL Overview

CPL is an extension of C++ implemented through a preprocessor. The preprocessor recognizes the special keywords listed in Table 1 and generates C++ code. Since CPL is compatible with C++ it includes the

C++ class relations such as inheritance and the use relation. Like C++, local, function, file and class scopes are defined in CPL. In addition to these; CPL uses program, class and object scopes. Since CPL is a distributed language maps active objects to distinct processes, conventional global variables are not applicable. Therefore, CPL class scope is strictly enforced. Using common variables for different CPL classes causes undefined results, as these variables are effective only in the same program context. Shared or global variables have to be implemented as CPL objects where other C++ scopes can be used in a conventional way only within an active object.

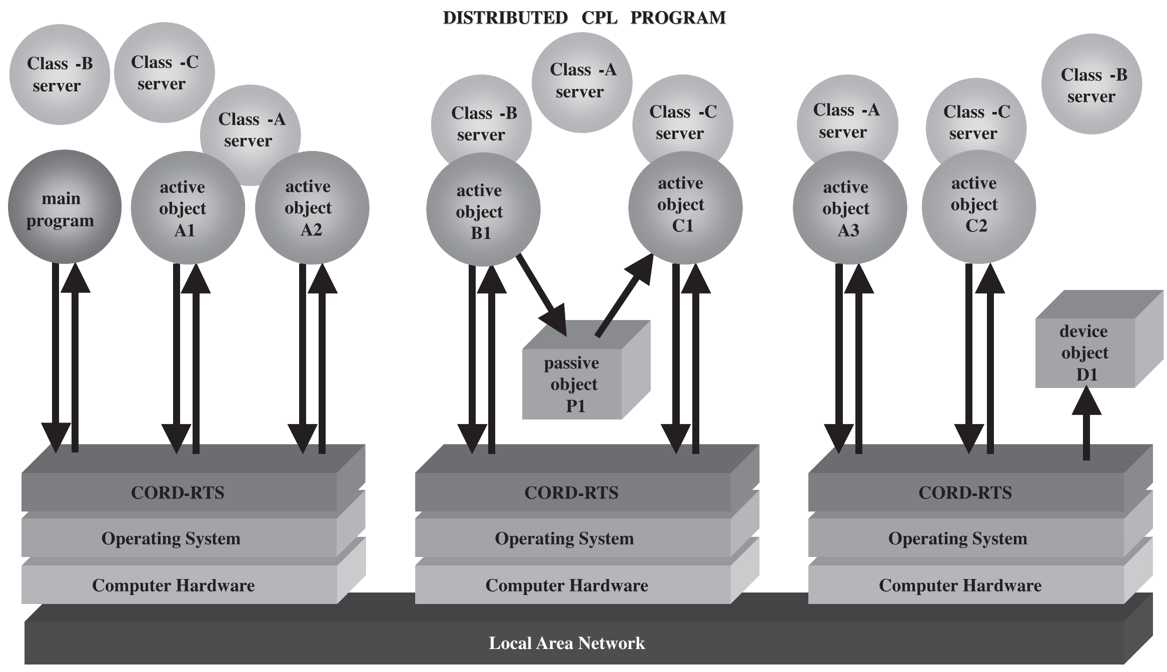


Figure 1. CPL Program Components

Table 1. CPL Keywords

active, passive, device	in, out, inout	buffer <B>
declare {...}	periodic <T>	speed <S>
inherits <C>	publish	do {...} every <T>
uses <C>	subscribe <M> to <C>.<N>	do {...} until <T>
includes <F>	body <C>	on "S"
priority <P>	class_main	at <T>
data	program <P>	timeout {...}
dynamic <B>	read <T>	retry <N> {...}
methods	write <T>	exception {...}

As a real-time distributed object language, CPL is characterized by the following features:

Concurrency: The unit of concurrency in CPL is an active real-time object. Active objects communicate with each other through method calls. CPL uses coarse granularity and does not allow additional concurrency within active objects. By using loosely coupled computers a maximum level of concurrency can be achieved.

Distribution: CPL programs execute on a distributed environment. Objects can be initiated on local or remote nodes. Active objects residing on different nodes communicate in exactly the same way as those residing on the same node without requiring location information. Passive objects are accessed by the local active objects in order to provide fast data exchange. Device objects can be accessed from any location, providing remote device control.

Communication: Communication is through uniform method invocation on both local and remote active objects. Method calls are either synchronous or asynchronous, depending on the return type of parameters specified in the method declaration in the class specification. Methods that return any kind of information are called synchronously, forcing the caller to wait until execution is complete. Communication between active and passive objects that reside on the same node does not initiate a remote method call; instead, a simple function call is used in the program context of an active object. In addition, the publish-subscribe mechanism for method multicasting is a faster mode of communication as its implementation does not require a method call, but delivers data when available.

Robustness: CPL has various error detection and exception handling mechanisms that can be used to develop fault-tolerant systems. Objects in a distributed system can be migrated to other nodes in case of failures. As a support system, CORD-RTS provides several fault- tolerant features.

Soft real-time: CPL supports soft real-time systems, which require response times in the order of milliseconds.

Hot code upgrade: CPL allows program code to be changed in a running system. It is possible to execute, replace, restart and move programs and objects on-line as long as all related units are re-compiled and re-executed.

5.1. Real-Time Features of CPL

CPL enables efficient specification and execution control of the following features to meet the *real-time requirements* of distributed object applications:

- Priority value assignment to active objects
- Specification of both priority and time constraints for method calls
- Deadline imposition for the arrival of results from the invoked objects
- Time referencing through a basic time type
- Timing accuracy maintained in the millisecond domain
- Execution of periodic and scheduled events, activated on a real-time basis
- Iterative computations with time constraints
- Fast and reliable access to shared data
- Exception handling for deadline violations

5.2. Distributable Software Units

While conventional C++ objects are non-distributable passive units, CPL introduces distributable active real-time objects along with passive objects and device objects.

5.2.1. Active Objects

Active real-time objects, which are instances of the CPL-defined *active classes,* are the primary processing units of a distributed CPL program. An active object is, in contrast to a passive object, an object that is associated with an independent flow of execution. It is capable of calling methods of all other types of objects and provides a set of service methods that can be called from outside. Each active class has a server that creates instances of active objects with the exact functionality defined in the class. Each node has a copy of each active class server, ready to create new objects that differ from each other only in the contents of their data members. Active real-time objects can be distributed over the network both at run- time and off-line. They are implemented through Unix processes and their scheduling is handled by the operating system of the node on which they reside. The class declaration indicates a scheduling priority, in the range of 1 to 10. This value is mapped to the underlying operating system scheduling scheme during object creation.

An active class is declared by using the keyword *active class class-name.* A class-name defines a class uniquely in the global scope of a distributed CPL program. The optional inheritance list follows the declaration. This list specifies CPL classes that are used for public inheritance. A derived class may inherit data and lists of methods, periodic functions, published methods and subscription lists from base classes. The basic structure of an active object consists of the following sections:

Static-data section: The keyword *data* introduces a list of static data members, which are kept inside the object context. The life cycle of these static data members depends on the life cycle of the active object. Their values are completely lost when the active object terminates. Each active object instance has its own set of data members.

Dynamic-data section: The dynamic data part of an active class provides a persistent data store facility and is kept in a safe storage inside the RTS. Since this storage has to be managed by the active object, its size, in bytes, has to be specified after the keyword *dynamic.* Although an active class may have any number of static data stores, it may only have one dynamic data store. The dynamic data storage is managed by the nodal RTS so that when an active object terminates abnormally, its dynamic part stays in the RTS until it is cleared. If an object terminates itself after all of its statements are executed, then it deallocates its dynamic data store on the RTS.

Service-methods section: Service methods that are listed after the keyword *methods* are the methods of an active object that can be called from other active objects. They do not return a value, but exchange information through parameter lists. Method parameters are specified by keywords *in*, *out* or *inout* to control communication. The keyword *in* declares a parameter that is received from the caller, while the keyword *out* specifies a parameter that is returned to the caller. If a parameter is accepted from the caller and is returned to it after modification then it is declared as an *inout* parameter. A call to a method that has only *in* parameters, or no parameter at all, results in an asynchronous and unidirectional call from the caller to the callee. If the method called has at least one *out* or one *inout* parameter, then the result is a synchronous and bi-directional call, which causes the caller to suspend execution and wait for a reply from the callee.

Periodic-methods section: Active objects can specify time-triggered method invocations at regular intervals in the millisecond domain. It is possible for an object instance to install any number of timers to trigger any number of events. The section marked with the keyword *periodic* and an integer value to represent a time interval in milliseconds indicates a list of parameterless methods that are to be called periodically at each interval.

Published-methods section: An object can produce data and publish it over the network. Published methods that are listed after the keyword *publish* are the methods of an active object that can take part in an inter-object communication where the target objects are not predetermined by the owner of the published method. The target objects are those that have subscribed to a particular published method, thus taking part in the target list of a multicast communication.

Subscribed-methods section: The keyword *subscribe* introduces a list of subscription declarations that specify lists of published methods the object is interested in and the particular local object methods they should trigger when published by other active objects. The subscription mechanism is handled transparently by the RTS system.

Class-body section: The class-body section is indicated with the keyword *body* and contains the method implementations with an optional *main* part. The main part may include any number of statements, which are executed only once, on object elaboration, after which the object starts to wait for incoming method invocations.

Following is CPL code for an active class, *Class_A*, which computes and publishes data periodically according to some input and reports its availability at certain intervals. The class has a number of data members, including a declaration of another active class instance. It also has a dynamic part of size 1024 bytes for persistent data. All object instances of this class possess the same methods and perform periodic processing at every 3000 and 1000 milliseconds. The class is capable of publishing *NewData* when *SendNewData* is activated every 3000 milliseconds. The class instances are triggered by the methods of device class *Class_D*, and active class *Class_B*.

```
device class Class_D;       //Obj_D exists in the system
active class Class_A {
uses Class_B, Class_D, SystemManager;
priority 5;
data:
  Name       own_name;       //a string-holder class
  int        mem_size, num_elem;
  int        current_value, value;
  Boolean_Type ready;
  SystemManager sys_mng;
dynamic 1024:
  Link_List element_list;
methods:
  GetNumOfElem(out:  int N);
  SetMemSize(in:  int S);
  Compute(inout:  int X);
  Calculate(in:  int Num, out:  int Y);
  Insert(in:  int Num);
  SetValue(in:  int Val);
periodic 3000:
  SendNewData();
periodic 1000:
  ReportHeartBeat();
publish:
  NewData(int D);
subscribe:
  SetValue to Class_D.Obj_D;
  Insert to Class_B.Produce;
};
// CLASS BODY IMPLEMENTATION
body Class_A {
GetNumOfElem(out:  int N)
{ N = num_elem; }
SetMemSize(in:  int S)
{ mem_size = S; }
Compute(inout:  int X)
{ X = X * value; }
Calculate(in:  int Num, out:  int Y)
{ Y = Num * Value; }
Insert(in:  int Num)
{ element_list.Insert(Num); }
SetValue(in:  int Val);
{ value = Val; }
SendNewData()
{ NewData(current_value); //publish current_value with this method }
ReportHeartBeat()
{ sys_mng.Report(own_name, STATUS_AVAILABLE); }
// NewData(int D) //no implementation is required for this method
class_main:  //This part is activated once during the elaboration
  cout << ''This is the class main.''  << endl;
  System_Manager sys_mng(''NODE_1'');
  sys_mng.Report(own_name, STATUS_AVAILABLE); //initial report
}
```

5.2.2. Passive Objects

Passive objects perform storage management of and provide fast access to the shared data store located on the main memory of a node that participates in a distributed application. A passive object is an instance of the CPL defined class *passive class*. The shared data store of a passive object is kept in safe storage inside RTS. Pointers and dynamic data structures whose sizes are not known at compile time cannot be declared as data members because these structures must be serialized contiguously. CPL does not automate mutual exclusion issues, as fast data access is favored. However it provides basic mechanisms that implement *CPL_LOCK* and *CPL_UNLOCK* primitives to control and serialize concurrent access to elements of shared data. Thus, the programmer can enforce mutual exclusion through these primitives if necessary. It should be noted that using these primitives increases data access from the nanosecond level to a few hundred microseconds.

Methods of a passive object can only be invoked by active objects residing on the same node; thus fast data sharing and explicit object and resource synchronization is provided. A passive object cannot issue calls to methods of other objects. The basic structure of a passive class consists of sections that specify an inheritance list, a shared-data store after the keyword *data*, data-store management methods listed after *methods* and their implementations in the class body marked with the keyword *body*.

A sample passive class declaration that can store indexed data is shown below. It has a shared-data store *Item*, an array, and two methods to access it, through an index.

```
passive class Class_P {
data:
  Item list[100];
methods:
  Insert(in:  int Value, in:  int Index);
  Get(in:  int Index, out:  int Value);
};
body Class_P {
Insert(in:  int Value) {
  CPL_LOCK;
  list[Index] = Value;
  CPL_UNLOCK;
}
Get(in:  int Index, out:  int Value) {
  CPL_LOCK;
  Value = list[Index];
  CPL_UNLOCK;
}
}
```

5.2.3. Device Objects

A device object provides a high level, standard and coherent interface to an input/output device installed on a node. Any active object located on any node can access a device object through the subscription mechanism. The Device Manager, as part of the CORD-RTS, manages subscription lists, registering active objects that are interested in a specific device data. Whenever data is available on the device, the manager reads the data into a local buffer together with a time stamp and issues calls to the registered methods of the active objects that are on the subscription list of the device object. Device classes cannot inherit or use other classes.

The structure of a device object, which is an instance of the CPL defined *device class*, specifies the access mode of the device (*read* or *write*) and the type of data, the speed of the device in bauds after the keyword *speed*, the capacity of an internal buffer to hold device data after the keyword *buffer* followed by the number of elements and the priority of the device object after the keyword *priority* and an integer from 1 to 10. The Device Manager keeps a data buffer of the specified type and access mode, and refreshes its contents whenever new data is made available. The implicit class section consists of methods acquired by default, depending on the access mode of the device. They are *Create(..), Bind(..), Delete(..), Read_Data(..), Write_Data(..)*, *Subscribe(..)* and *Cancel_Subscription(..)* with appropriate parameters. A device object publishes data whenever new data is read from the device. This feature is activated by active objects through the *Subscribe_Device_Data(..)* method.

Active objects may also read data either from the buffer or from the device at any time. This feature is specified as a parameter during the call to the *Read_Data* primitive indicating the age of the data in milliseconds. If the specified time value is greater than the current age of the data in the buffer, then the content of the buffer is returned. Otherwise, a new read attempt takes place. If a device provides data continuously then the Device Manager performs frequent read operations and keeps its buffer fresh at all times.

A sample device class declaration is shown below. It can read an integer from a device with priority 6 and a speed of 2400 bauds. It can hold 10 elements in its internal buffer.

```
device class Serial_Comm {
priority 6;
read int;
buffer 10;
speed 2400;
};
```

5.3. Object Manipulation

Object manipulating actions utilize all or a subset of the following default methods inherited by all CPL classes: *Create(..)*, *Bind(..)*, *Delete(..)*, *Move(..)*. These methods are defined by the CPL Preprocessor and their definitions take place in the client stub header and body.

Scope: CPL classes are declared in the global scope of a distributed program; therefore object instances of basic CPL classes (active, passive and device) are created global to all nodes. However, CPL classes use the same visibility rules as C++ and the CPL compiler manages scope rules considering files, functions and blocks. Thus, even if an object is created in global scope, it may still be inaccessible unless the block in which the declaration takes place is visible by the user object.

Naming: A distributed application must be capable of adapting to dynamic changes in network configurations. Objects that make up such applications need to be created anywhere on the network and be migrated across node boundaries, if necessary. This means that each object and each of its methods must have a unique, logical, system-wide name so that subsequent accesses to these objects do not require any location information. The RTS is capable of locating an active real-time object that corresponds to the symbolic name provided by the caller object. This feature is supported by a special four-level naming convention that all CPL objects use. Of these, the first level uses the program identification. The second level is the class type that is transparent to the user. The third level is the class identification, and the fourth level is the object identification. In order to speed up processing, numeric representations of logical names are used within the system. This four-level naming convention allows programmers to use the same names for different classes and objects within the same CPL program. It is also possible to run multiple copies of the same CPL program simultaneously on the same physical network.

Creation: CPL objects can be created on any node within the network. CPL supports both static declaration (class name and variable name) and dynamic allocation (class name, variable name and the *new* operator). Location information can be provided either as a constructor parameter or as a string proceeded by the keyword *on*, specifying the particular node on which the object is to be created. The following examples show program statements that create instances of classes.

```
#define NODE_1   105.23.25.01
#define NODE_2   105.23.25.02
Class_A    obj1 on ‘‘NODE_1’’;
Class_A*   obj2 = new Class_A; //on the current node
```

```
Class_A    obj3(initial_value) on ''NODE_2'';
Class_A*   obj4 = new Class_A on ''NODE_1'';
Class_A*   obj5 = new Class_A(''NODE_2'');
Class_A*   obj6 = new Class_A(the_node); //the_node is a variable
                                         //determined at run-time
Class_A*   obj7 = new Class_A(initial_value) on ''NODE_1'';
Class_P*   obj8 = new Class_P(initial_value); //passive object
Class_D*   obj9 = new Class_D(''ttya'') on ''NODE_2''; //device object
```

An active class may possess any number of user-defined constructors in addition to the default constructor. The default constructor creates an object reference to which an existing object has to be bound through the *Bind()* function. Objects can be created using the *Create()* function of the object reference.

```
Class_A actual_object;  //data member of an active class
actual_object.Create(); //object instance is explicitly created
Class_A actual_object; //data member of another active class
//use one of the following to bind to the already created object
actual_object.Bind("actual_object");
actual_object.Bind(actual_object.Get_Name());
```

When a constructor with parameters is called by an active object, it creates a new active object with the object name and initializes it with the provided parameters. Only *in* type of parameters are allowed in a constructor. The following are examples of class constructors:

```
Class_A();
Class_A(in:  int Val);
Class_A(in:  int X, in:  float Y);
```

Deletion: A CPL object is destroyed when its default method *Delete()* is invoked by an active object or by the main module of the program. This call results in the location of the node on which the object resides, the deallocation of its resources and the removal of the object representation from the system.

Relocation: A call to the default method *Move()* specifying the destination node results in the migration of the object. Relocation is performed by stopping the object process and recreating it on the specified target node. Meanwhile, the Object Manager forwards incoming messages by changing the message destination to the new location. If a dynamic data store takes part in the object, its content is entirely copied to the new location, thus preserving the persistence of the object state. The new object can then retrieve its previous data or start over.

5.4. Interaction Among Objects

A practical real-time object-oriented language must support multiple types of communication constructs. Communication in CPL is either through point-to-point operations, which involve a single source and a single destination object, or collective operations in which more than two objects participate. Both types of communication take place through high level constructs, namely, object method calls. The CPL compiler translates method calls into appropriate low-level communication primitives, which involve method invocations on either local or remote objects. Communication is transparent at the programming language level where there is no reference to the location of objects, thus providing the programmer with a single object space. This feature is realized through CORD-RTS mechanisms that make object references meaningful across node boundaries. All method calls and replies are converted into system level messages having priority values, source, destination, and data parts with time stamps.

5.4.1. Synchronous Communication

In a synchronous call, a CPL client object that issues a method call blocks and waits until the method of the destination object returns a result. A call to a method that possesses at least one out or *inout* type of parameter initiates a synchronous communication. RTS suspends the execution of the caller object until the callee returns a value determined by the *out* or *inout* typed parameter. An *out* type of parameter causes a unidirectional data exchange from the callee to the caller while an *inout* type of parameter initiates a bi-directional data exchange between the participants of the communication.

5.4.2. Asynchronous Communication

In an asynchronous call, a CPL client object initiates a communication request and immediately proceeds on its execution path without waiting for the call to proceed and terminate. A call to a method with only *in* type parameters causes an asynchronous communication, where a one-way data exchange takes place from the caller to the callee. A call to a method with no parameters also causes an asynchronous communication and serves the purpose of synchronization between objects as no data exchange occurs.

For example, a server object instance of class *Class_A* declares the following methods:

```
Get_Value(out: int Val)
Set_Value(in: int Val)
Compute(in: int X, out: int Y)
Extrapolate(inout: int X, in: int Y)
```

following method calls are examples of synchronous/asynchronous communication for the above declarations:

```
obj1.Set_Value(value);                //asynchronous
obj1.Compute(value, result);          //synchronous
obj1.Extrapolate(old_value, new_value); //synchronous
```

5.4.3. Multicast Communication

In addition to the interaction mode based on point-to-point operations, CPL provides another interaction mode in which more than two objects take part. This is multicast communication where a source object delivers copies of a single message to each object on a specified list of destinations without holding an explicit reference to each one. The source object is either an active object or a device object that publishes a method m, and objects on the destination list are those objects that have subscribed to that particular method m.

An active class can publish a method either periodically or when a certain event occurs. In the first case, a periodic method generates calls to a published method. In the second case, the call to the published method is generated by a service method, on the occurrence of an event. A published method is actually a declaration of information that is to be distributed. The parameter list of the method specifies the data type of the information and the method itself requires no implementation.

```
publish:
  New_Data(int v1, int v2, float v3, char v4);
  Update(int v1, float v2);
```

An object is interested in a published method subscribes to that method. The keywords *subscribe - to* introduce subscription declarations, each specifying a pair of methods: the published method (through class name and method name) and the triggered method, a local method of the subscribing object is to be invoked when a communication message generated by a published method arrives. RTS handles the publish-subscribe mechanism and issues calls to the related methods of the subscriber objects.

```
subscribe:
  Set_Value to Class_C.New_Data;
```

An active object can also subscribe to a device object, which publishes data whenever it is available. In this case, there is no specific method, but the device class name and its data type are used. The Device Manager handles the necessary actions.

Strong typing is enforced for the subscription mechanism. The parameters of the triggered method of the subscribing object have to match the parameters of the published method. A method may be involved in more than one subscription, but published methods cannot be overloaded. An active object can cancel its subscription at runtime.

5.5. Time-Related Constructs

Time-related constructs are primitives that supervise timed events. They are used to express the timing behavior of an application as the timeliness of a response to an event. All time- related constructs take part in active object implementations since an active real-time object is the basic unit of execution of a CPL program.

CPL introduces a new basic type, *Clock_Time*, recognized in code by the CPL Preprocessor, to denote daily time in 24-hour format in millisecond resolution. A time constant is in the format *hh:mm:ss.mmm*.

5.5.1. Periodic Methods

An active object can specify time-triggered method invocation at regular intervals in the millisecond domain in its periodic-methods section. These methods map to individual POSIX-compliant threads that are activated by the system timer. The actual invocation time for periodic timers is calculated continuously according to the following algorithm:

```
start_time = Get_Current_Time();
next_time = start_time + period;
sleep_time = period;
LOOP
  Sleep(sleep_time);
  //*** Do periodic processing here ***//
  current_time = Get_Current_Time();
  next_time = next_time + period;
  sleep_time = next_time - current_time;
END LOOP
```

An active object declares periodicity with the keyword *periodic* followed by an integer value that represents a time interval in milliseconds. This construct is followed by a list of parameterless methods that are to be invoked periodically at each time interval. An active object may possess any number of periodic methods, each managed by distinct timers.

```
periodic 3000:
  Send_Report_A();
```

```
    Send_Report_C();
  periodic 1000:
    Log_Data();
```

Each method in a periodic section installs a separate timer with the indicated expire time starting when the object is elaborated. In order to preserve accurate periodicity, each timer keeps track of absolute time, rather than the interval between successful calls. Thus, the time it takes to execute a method is included in the interval. The result is undetermined if the time to execute a method is more than that of the specified interval. If more than one timer expires at the same time, each triggered method is actually executed sequentially, in an order determined by the scheduler. CPL does not enforce any priority mechanism for timers.

Figure 2 illustrates the order of execution with respect to time frame for an example period of 500 milliseconds. The figure also shows possible reasons that may cause time shifts due to process and thread scheduling and non-atomic calculations for the *next_time* value.

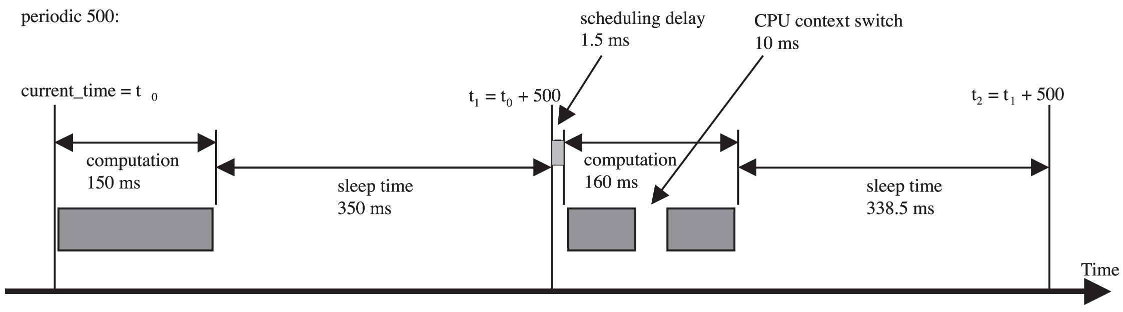


Figure 2. Order of Execution

5.5.2. Timed Loops

CPL has extended regular C++ iteration statements with timing assertions, making it possible to explicitly express timing requirements on object behavior. Two types of timers are used in their implementation: one-shot timers and periodic timers. A one-shot timer fires off once at the specified time. If the current time is later than the specified time, the system takes action immediately. A periodic timer fires off at specified intervals. The precision of timing depends on the priority driven scheduler of the underlying operating system because, when a timer expires, the exact moment the triggered action executes is determined by its priority value.

do-every loop: This iteration structure starts with the keyword *do* and repeats itself at time intervals specified after the keyword *every*. The iteration continues until the loop is explicitly broken. Otherwise, the next statement is never reached.

```
do {
  Log.Write_Data();
} every  1000;
```

The specified time value is in milliseconds and the implicit sleep time after the execution of one iteration is computed at each cycle in order to set up the timer at exact intervals. The result is undefined if the time needed to execute the body of the iteration is higher than the specified interval.

do-until loop: This structure executes iteration starting with *do* until the current time exceeds the specified wall clock time after the keyword *until*. The time value is specified after the keyword until, in 24-hour format (hh:mm:ss.mmm). Any setting exceeding a day has to be handled by additional algorithms.

There may be a blocking function call such as *Sleep()* inside the loop.

```
do {
  obj1.Read(Value);
  obj1.Write(Value);
  CORD_RTS.Sleep(1000);
} until 20:45:00.00;
```

Iteration starts immediately and repeats itself continuously unless broken by a break statement. The current time is compared with the specified time at the end of each iteration. If it is earlier than the specified time, another iteration starts. If the specified time expires during the execution of the loop body, it is completed before the iteration terminates. Thus, in some cases, the specified time may not be the exact time when the iteration ends.

5.5.3. Timed Statements

The keyword *at* is used to express timing requirements at the statement level. The *at* primitive blocks the execution of a statement or a block to which it is appended until the wall-clock time reaches the value it specifies, thus enabling programmers to determine in advance when certain actions will execute. The blocked program flow resumes execution when the specified clock-time is reached.

```
obj1.Set(Val) at 15:30:00.0; //initiate method call at 15:30:00.0
value = 100 at 12:00:00.0;   //assignment action takes place
                             //at 12:00:00.0
{ value = 200;
  Process_Data(value);
} at 12:30:00.0; //execute block of statements at time 12:30:00.0
```

5.5.4. Exception Handling

CPL supports implicit exception control for method calls. The keywords *exception*, timeout and *retry* are used to define handler blocks. When an exception is raised, control is passed to the statements inside the handler block.

Exception Block: This is the most general form of exception handling. If any error occurs during the processing of a synchronous method call, exceptions are caught and the statements in the block following the keyword exception are executed.

```
obj1.Get_Data(data, CORD_PRIORITY_3, 500) exception
                                    { Display_Report("Error"); }
```

Timeout Block: CPL allows the specification of a deadline for returning results in a synchronous method call. Each manager in the RTS receives this deadline value as one of the parameters associated with the method call and checks whether or not the results come back within the specified deadline. If not, it does not process the request message and sends a system reply to the sender. The sender object then raises an exception that is handled by the timeout block that follows the method call. The keyword *timeout* introduces the timeout block to which control passes after a timeout exception is raised.

```
obj1.Get_Data(data, CORD_PRIORITY_3, 100) timeout { Alert(); }
```

In this example, the method has a priority of value 3 and the deadline for result return is 100 milliseconds after calling time. If timeout occurs before the reply is received, the statements in the timeout block are executed. If a call does not specify a timeout value, then the system default priority (five) and an infinite time interval is assumed.

The RTS checks the result of the following expression to determine if a message has the potential to complete in its specified deadline and, thus, is worth processing.

```
(message_time_of_initiation + message_timeout -
  (TOTAL_COMMUNICATION_DELAY + Cord_Time.Get_Current_Time) ) <= 0
```

Total communication delay is a system parameter, which is previously computed for a network system considering average latency for a message to be transmitted from one node to another. This value may also be periodically measured to obtain a dynamically changing average value.

Retry Block: CPL allows a method call to be repeated a number of times in case an error prevents it from completing successfully before an exception is raised. The keyword *retry* following the method call introduces the repeat count and an exception-handling block. If the method call is successful, execution continues with the next statement. In case an error of any kind occurs, an implicit counter is incremented and the call is issued again if the number of trials denoted by retry has not been reached yet. Otherwise, control passes to the statements in the retry block.

```
obj1.Set_Value(65, CORD_PRIORITY_4, 1000) retry 5 {Report_Error();}
```

The difference between a timeout and retry type of exception handling is that the first one catches only timeout exceptions, while the second catches all types of exceptions raised during a call after a number of unsuccessful trials.

6. Application Development With CPL

The process of application development in CPL has two phases: program development and system generation. The result of the first phase is a collection of modules that constitute a logical solution to the problem. During the second development phase, the program modules are combined and translated into an executable system.

A CPL program consists of one or more CPL modules, classes and their bodies. CPL modules are compiled with the CPL compiler, which includes a C preprocessor, the CPL Preprocessor and a C++ compiler. Compiling the main module results in a header and body file, which are then linked with appropriate files to form a single executable. Compiling an active class produces client stub code as a header and a body file, a server header and a body file. After linking, the server files construct an executable. Compiling a passive or a device class produces header and body files, which are to be linked to an executable, such as an active class or the main module. Figure 3 illustrates a standard application development process.

When a CPL program is loaded onto the RTS, first its main module is elaborated. It initially registers itself and all of its classes and then waits for the start command. After the start command is given by the user via the command shell, the program starts executing the first line of its code right after the keyword program.

Termination of the program can be controlled either explicitly or implicitly. A call to the *Exit()* function of the CORD Interface explicitly terminates the program. Implicit termination occurs after the last line of code in the program is executed. Another way of program termination is to send a stop command via the command shell. In any case, termination results in the deletion of all objects, and the unregistering of all classes and the program itself. Figure 4 illustrates the loading and execution phases of a CPL application.

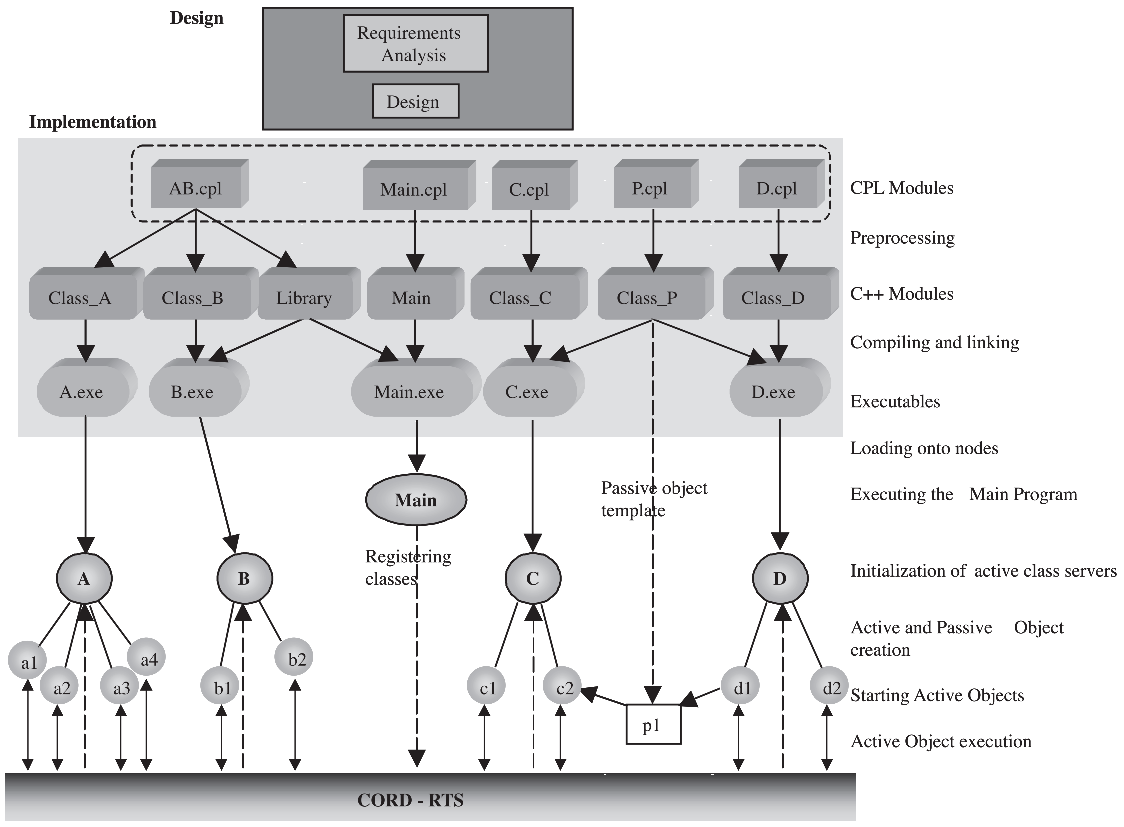


Figure 3. Development Process

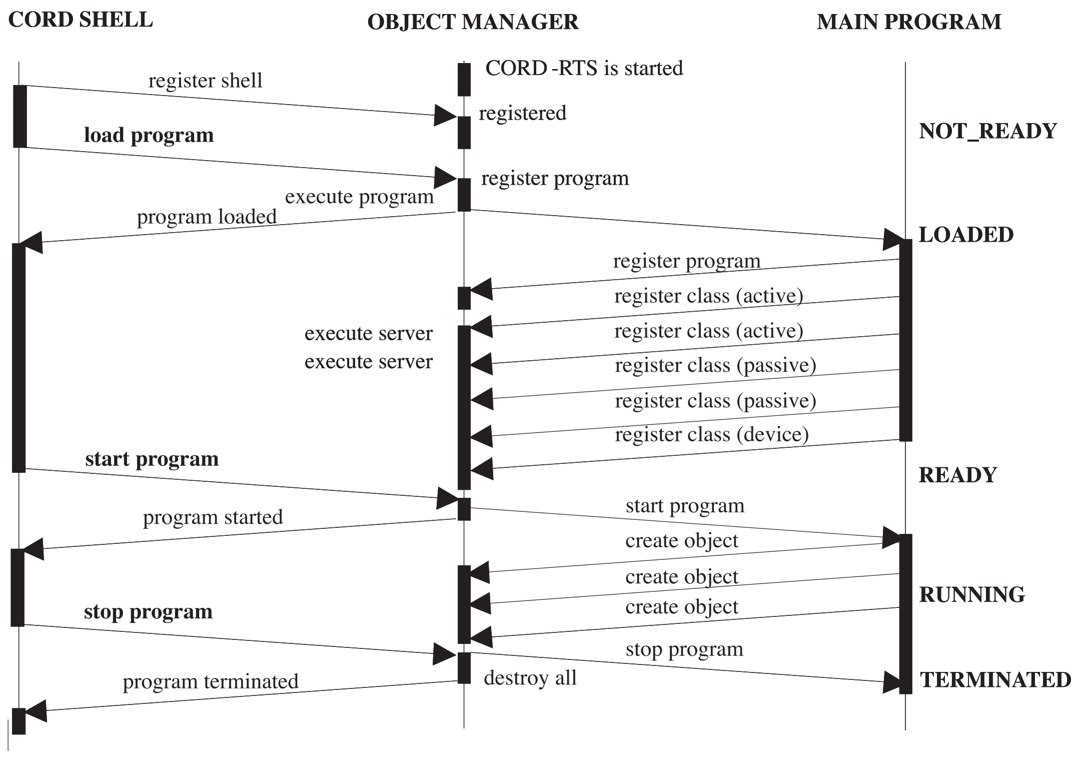


Figure 4. CPL Program Execution Steps

7. Sample Application

A simple distributed application is illustrated in Figure 5. This program computes the squares of the numbers from 1 to 10 and displays the results on the screen. The main module `TakeSquare` is executed on `NODE_1`, which creates the active object `value_display` and the passive object `value_store` on the same node, and another active object `value_square` on `NODE_2`. It makes a call to `value_square` iteratively with a number in the list and receives its square, writes this value to `value_store` with an index, and then calls a method of `value_display` with the index to be read and displayed. This program is fed to the CPL Preprocessor and resultant C++ code is compiled and linked with the provided makefile, producing three executables, one for the main module and two for the active class servers.

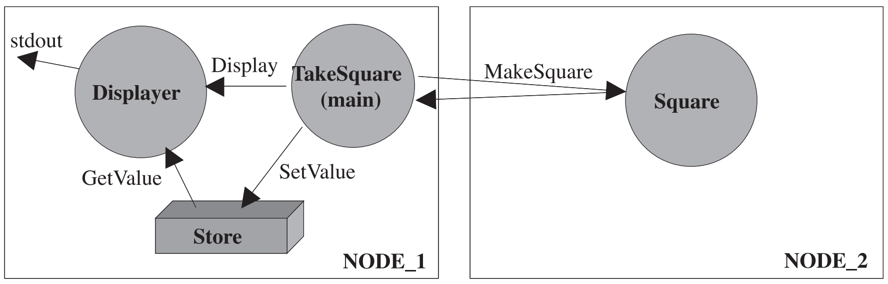


Figure 5. Application Program Taking Squares

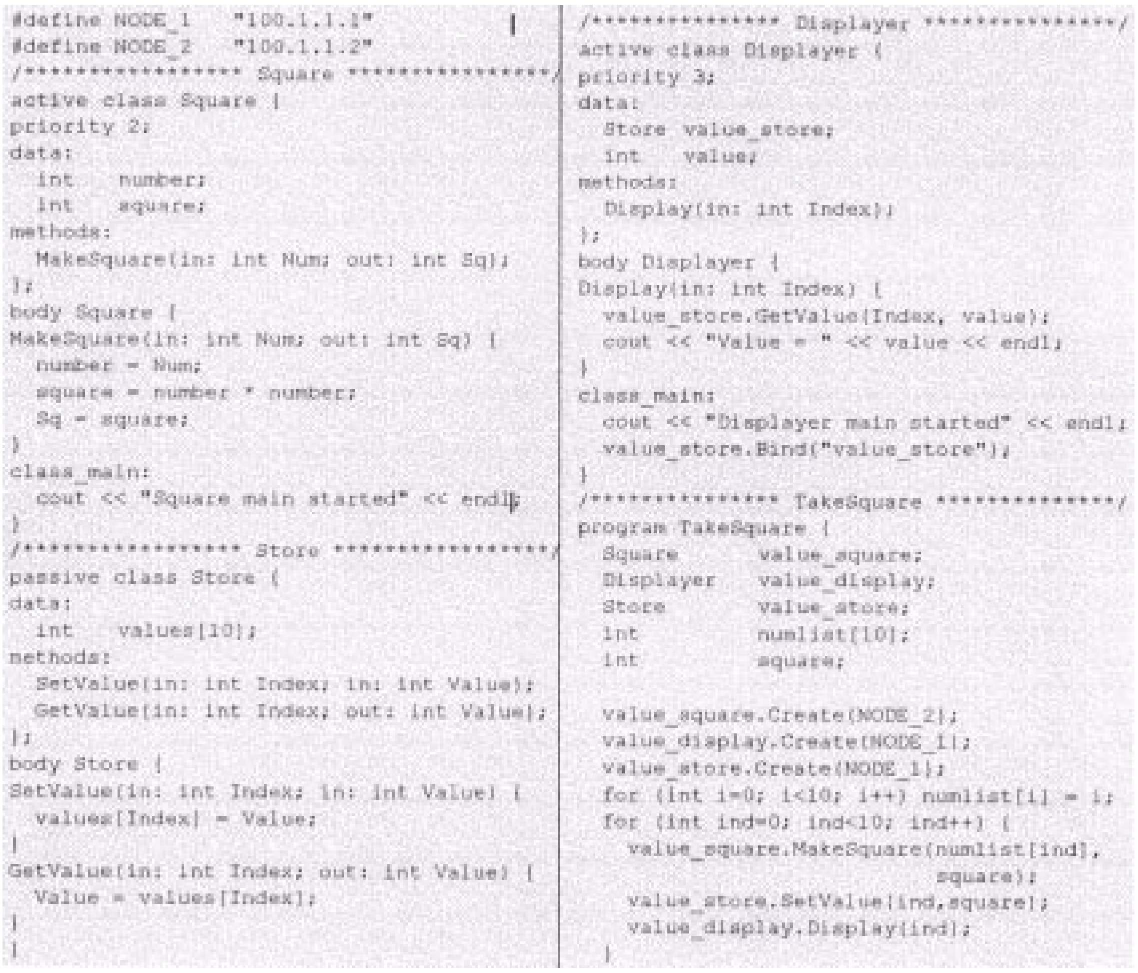
The source code in CPL implementing this distributed program is given in Figure 6.

8. Conclusion

CPL and its run-time support system CORD-RTS provide a high-level, real-time distributed object-oriented programming environment. It is based on a new object model, the active real-time object with message and time initiated methods. CPL has focused on the ability to express distribution issues and real-time constraints through language constructs to simplify program design. Thus, programmers can concentrate on functional behavior rather than complex, low-level implementation issues.

CPL enables efficient specification and execution control of several new real-time features. These contributions are based primarily on the requirements of distributed command and control systems, but serve as well the needs of a large class of distributed applications with soft real-time requirements, such as distributed knowledge based systems or parallel computation. Encapsulation of activity, data and device access into objects is a novelty of CPL that allows for the easy and reliable development of complete systems. Publish/subscribe mechanism, synchronous, asynchronous and multicast type of communication, timed statements, deadline specification and real-time execution control capabilities are features that make CPL distinct in a class of languages that address similar objectives.

It is possible to make enhancements to the CORD System to make it more portable and robust. Currently, keeping the language intact, the system can be ported to other platforms by just modifying the related parts of the RTS. In the future, it may also be possible to make use of CORBA [11] and Adaptive Communication Environment (ACE) [13] to replace some operating system dependent parts of the RTS for higher portability. The current CPL Preprocessor is also open to improvements. A more sophisticated error handling mechanism may be implemented. Although the syntax checking is sufficient, semantic analysis can also be improved.



```
#define NODE_1   "100.1.1.1"
#define NODE_2   "100.1.1.2"
/***************** Square ****************/
active class Square {
priority 2;
data:
  int   number;
  int   square;
methods:
  MakeSquare(in: int Num; out: int Sq);
};
body Square {
MakeSquare(in: int Num; out: int Sq) {
  number = Num;
  square = number * number;
  Sq = square;
}
class_main:
  cout << "Square main started" << endl;
}
/***************** Store *****************/
passive class Store {
data:
  int   values[10];
methods:
  SetValue(in: int Index; in: int Value);
  GetValue(in: int Index; out: int Value);
};
body Store {
SetValue(in: int Index; in: int Value) {
  values[Index] = Value;
}
GetValue(in: int Index; out: int Value) {
  Value = values[Index];
}
}
```

```
/*************** Displayer ***************/
active class Displayer {
priority 3;
data:
  Store value_store;
  int   value;
methods:
  Display(in: int Index);
};
body Displayer {
Display(in: int Index) {
  value_store.GetValue(Index, value);
  cout << "Value = " << value << endl;
}
class_main:
  cout << "Displayer main started" << endl;
  value_store.Bind("value_store");
}
/*************** TakeSquare **************/
program TakeSquare {
  Square      value_square;
  Displayer   value_display;
  Store       value_store;
  int         numlist[10];
  int         square;

  value_square.Create(NODE_2);
  value_display.Create(NODE_1);
  value_store.Create(NODE_1);
  for (int i=0; i<10; i++) numlist[i] = i;
  for (int ind=0; ind<10; ind++) {
    value_square.MakeSquare(numlist[ind],
                            square);
    value_store.SetValue(ind,square);
    value_display.Display(ind);
  }
```

Figure 6. Sample CPL Program Code

CPL has been implemented at the Turkish Navy Software Development Center. New features are being added as intensive tests are performed using a simple prototype DC^2 system.

References

[1] D.S. Reeves, K.G. Shin, "Parallel and Distributed Real-Time Computing", IEEE Parallel & Distributed Technology, Vol.2, No. 4, pp. 8, 1994.

[2] B. Selic, "A Generic Framework for Modelling Resources with UML", Computer, pp. 64-69, June 2000.

[3] E. Saridogan, "Design and Implementation of a Concurrent, Object-Oriented, Real- Time and Distributed Programming Language with its Supportive Run-Time System", Ph.D. Thesis, Istanbul Technical University, Institute of Science and Technology, January 2000.

[4] E. Saridogan, N. Erdogan, "A Real-Time and Distributed System with Programming Language Abstraction", International Conference on Parallel and Distributed Processing Techniques and Applications, June 28-July 1, Las Vegas, USA, 1999.

[5] A.S. Grimshaw, "Easy-to-Use Object Oriented Parallel Processing with Mentat", IEEE Computer, May 1993.

[6] Y. Ishikawa, H. Tokuda, C.W. Mercer, "An Object-Oriented Real-Time Programming Language", Computer, October 1992.

[7] Y. Yokote, "The Design and Implementation of Concurrent Smalltalk", World Scientific, Vol.21, 1990.

[8] B. Wyatt, K. Kavi, S. Hufnagel, "Parallelism in Object-Oriented Languages: A Survey", IEEE Software, Nov. 1992.

[9] P.G. Robinson, J.D. Arthur, "Distributed Process Creation Within a Shared Data Space Framework", Software-Practice & Experience, Vol.25(2), February 1995.

[10] P. America, "POOL-T: A Parallel Object-Oriented Programming", Research Directions in Object-Oriented Programming, B.D.Shriver, P.Wegner, MIT Press, Cambridge, Mass. 1987.

[11] K. Smith, A. Chatterjee, "A C++ Environment for Distributed Application Execution", Tech.Report ACT-ESP-275-90, Micro-electronics Computer Technology Corp., Austin, Texas 1990.

[12] A.D. Stoyenko, W.A. Halang, "Extending Pearl for Industrial Real-Time Applications", IEEE Software, July 1993.

[13] D. Kirshen, "An Overview of the Mach Operating System", Operating Systems Technical Committee Newslett-ter, 3(2), 1989.

[14] Object Management Group, The Common Object Request Broker: Architecture and Specification Rev.2.1, August 1997.

[15] G. Bollella, J. Gosling, "The Real-Time Specification for Java", Computer, pp. 47- 54, June 2000.

[16] D.C. Schmidt, F. Kuhns, "An Overview of the CORBA Specification", Computer, pp.56-63, June 2000.

[17] K.H. Kim, "Object-Oriented Real-Time Distributed Programming and Supportive Middleware", Proc. ICPADS 2000, Japan, pp. 10-20, July 2000.

[18] E. Saridogan, N. Erdogan, "An Efficient Middleware Architecture Supporting Real- Time Distributed Object Programming", Elektrik, TUBITAK, Vol. 10, No. 1, pp. 23-39, 2002.

[19] RFC 1305 - 1992, Network Time Protocol (V3), IETF.

[20] S. Ren, G.A. Agha, "RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems", ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, June 1995.

[21] D. Schmidt, December 1993, June 1994, "The Adaptive Communication Environment: Object-Oriented Net-work Programming Components for Developing Client/Server Applications", 11th and 12th Sun Users Group Conference.