The Clouds/Distributed Eiffel Programming Model

L. Gunaseelan and Richard J. LeBlanc, Jr.
College of Computing, Georgia Institute of Technology

1 Clouds Programming Model

Clouds [DCM+90, DLA88] is an object based operating system that provides direct OS level support for large-grained objects. The programming model that we discuss in this section is the one that is directly supported by the operating system. This model only provides a bare-bones support for object programming, and as discussed later in the section, is far from practical for realistic applications.

The Clouds programming model is based on two basic notions: *objects* and *threads*.

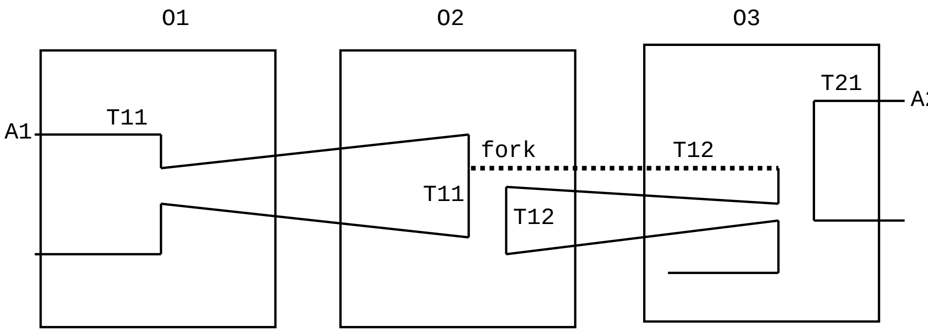

Figure 1: Clouds Programming Model

While the traditional notion of *objects* is to consider them as pieces of memory, Clouds advances the notion of objects to include entire address spaces. While other object-based systems map objects into an address space, in Clouds, objects *are* address spaces. These address spaces are *large-grained, passive, persistent* and *mobile.*

- *Large-granularity* means that the size of the object can be very large (of the order of several megabytes). Objects contain both code and data. Only the code within the object can manipulate the data there. The operating system enforces such encapsulation. It is easy to think of an object as the memory image of a typical Unix program.
- The objects are *passive* and do not have any built-in process attached to them. Specifically, objects do not include active server processes waiting to serve client requests.
- The *persistent* nature of the objects means that object state persists across computations.
- The objects (address spaces) can move between nodes at runtime, under programmer direction. The DSM subsystem in the Clouds OS can make an object available for invocation on any node. Thus, objects are both *units of distribution* and *mobility* in the Clouds model.

An *application* is built around a collection of such passive, persistent, mobile address spaces.

A *thread* provides computational activity in Clouds objects. A thread is a dynamic entity that refers to the flow of control of a computation. A thread can move across a collection of Clouds objects. When a thread is active in one object, it executes the code inside it and can manipulate the data in the object. A thread moves from one object to another by means of *invocations*.

A thread can fork new threads. A set of threads may share an address space. Figure 1 shows two applications A1 and A2 active in objects O1, O2 and O3. Application A1 consists of threads T11 and T12, with T11 forking T12. Application A2 consists of thread T21.

To summarize, Clouds objects are large-grained, passive, persistent address spaces that serve as both units of distribution and mobility. An object can invoke another by means of a remote procedure call (RPC), which results in the executing thread changing address spaces. Threads are dynamic entities that move across a system of objects. Starting a computation is starting a top-level thread in one of the objects, which may fork child threads, and may move across a set of objects. An application is structured around a collection of such persistent address spaces and may consists of multiple threads. Objects of an application may reside on the same or different nodes. The model of the entire *distributed system* itself, therefore, consists of a system of potentially sharable address space of objects with active threads in them.

It is obvious that a number of things are missing from the programming model, to make it a useful/practical programming system:

- The most glaring feature missing is the support for small objects. Every object is construed to be a large-grained address space and in a practical, performance oriented system, it does not make sense to represent every array, string or stack as a large-grained object. (Imagine performing an object to object invocation to access the next character in a string or the next element in an array). Also, since large-grained objects are known across a unique system-wide name space in the entire distributed system, imagine the growth in the name space if every fine-grained entity is represented as a Clouds object. Clearly, large-grained object representation is not appropriate for fine-grained entities. One important contribution of the Distributed Eiffel language, discussed in the next section, is the support for such fine-grained objects.

- The second type of support needed is the support for *parameter passing*. When a thread moves from one object to another through invocation, it carries parameters. How does the programmer pass parameters and what type of parameter passing semantics is appropriate?

- The third feature that is missing from the model is the support for concurrency control. The model leaves the control of concurrency and synchronization entirely up to the programmer. The Distributed Eiffel language, discussed in the next section, enforces a specific model of concurrency control by attaching semantics to the operations as *readers* and *writers*.

- The fourth type of support needed consists of facilities for a programmer to distribute/spread objects and threads across different nodes, allowing the programmer to dynamically control the placement of both objects and threads.

The Distributed Eiffel language extends the programming model by adding support precisely in these four different areas.

2 Distributed Eiffel Programming Model

Distributed Eiffel [GL92] is an extension of the Eiffel programming language. The Eiffel language [Mey88, Inc] provides support for fine-grained objects. It is a small language that combines a number of concepts into a cohesive set of features that are used to define a single syntactic construct called a *class*. The entire language design centers around how to define one *class*. Classes are templates for describing run-time object instances. Eiffel combines the notions of "module", "class", and "type" into one: a class is a module, and a class is also a type. With distribution in mind, if we use the same classes to define large-grained objects, then a class is also a unit of distribution, a shared address space, unit of mobility, and an entity that persists. Thus everything fits into a unified goal: support for one entity, class - provide the programmer with facilities for defining one class, and we have the ability to define and program the whole distributed system.

The Distributed Eiffel language is based on a two-tier granularity of objects: large-grained and fine-grained. Though a uniform model of objects that supports objects of single granularity would have been semantically more appealing, as we show in a later section, this two-tier classification is essential for practical distributed systems programming.

A typical Eiffel program is based on a dynamic model of fine-grained objects. Figure 2 shows the code for a BOOK object, and it's run-time representation.

```
Class BOOK
export issue_to, title, author, ...
feature
    title:STRING;
    catalog_no:INTEGER;
    author:PERSON;
    issued:BOOLEAN;
    ...
    issue_to (member:PERSON) is
        do
            ...
        end;
    ...
end -- class BOOK
```

```
class PERSON
export pname, year, ...
feature
    pname:STRING;
    year:INTEGER;
    Create (nm:STRING; yr:INTEGER) is
        do
            ...
        end
end -- class PERSON
```

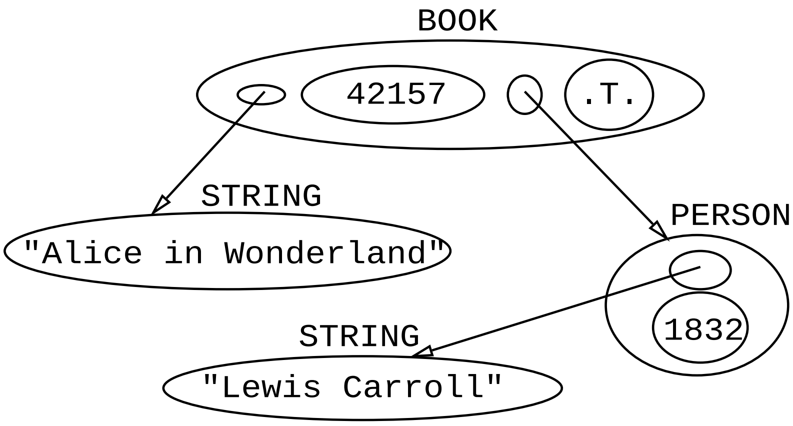

Figure 2: Eiffel Programming Model

The Eiffel language defines the notions of *basic types* and *references*. Specifically, a variable x

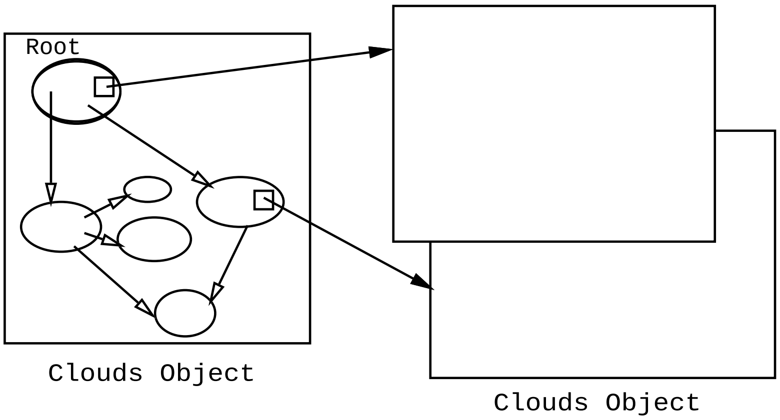

Figure 3: Distributed Eiffel Computation Model

in a program may be either one of the basic types (*integer, real, character* or *boolean*) or a reference to another fine-grained object. In the class BOOK above, attributes *catalog_no* and *issued* are of basic types integer and boolean respectively. Attributes *author* and *title* are references to other fine-grained objects. Thus, the basic object model consists of a graph structure of fine-grained objects, with each object having references to others. This model is very dynamic, because these fine-grained objects may get created on-the-fly, and may quickly become garbage. The execution of a sequential Eiffel program starts at a specific *root* object in its *create* routine and ends when the body of the *create* routine completes execution. Thus, the memory image of a sequential Eiffel program is a *cluster* of fine-grained objects in one address space, with a specific root object.

The extension to the model, namely the Distributed Eiffel language, enables several such clusters residing on different machines to refer to each other and invoke each other. Each such cluster resides in an address space of its own, with a well defined root object, with the entry points of the root object being the entry points of the cluster. As shown in Figure 3, each cluster corresponds to a large-grained object, with the entry points of the root object of the cluster becoming the entry points of the large-grained object.

Any fine-grained object in the cluster (including the root) can refer to any other cluster by means of variables that are declared with a slightly different syntax. An example is given in Figure 4. An important point to note is that a cluster (large-grained object) provides a higher level abstraction of the behavior of a group of fine-grained objects, with the cluster behavior abstracted into the entry-points of the root. Considered another way, each cluster is a just the root object, providing a set of services and in the process making use of a number of fine-grained objects. Thus, we feel the power of the language comes from the ability to model lower level abstractions into fine-grained objects and the higher level abstractions into a cluster, namely the large-grained objects. The real power comes from the fact that these clusters can invoke each other (that is, make use of the services of each other) and at the same time reside on different nodes. Moreover, these object clusters are persistent.

A variety of software modeling approaches can be easily implemented with this model. Each large-grained object can be seen as a passive server that provides a service, that any other active server (objects with threads in them) can invoke and make use of. Or one can look at the these persistent object clusters residing on different nodes as a real distributed data base, organized or distributed according to geographical and application grouping, with every programmer in the

```
Clouds class MAILER
export send_mail, deliver, read_mail, add_user, ...
feature
    site_id:INTEGER;
    site_name:STRING;
    user_table: ARRAY[STRING];
    remotemailer :: MAILER; -- reference to another instance

    send_mail (user_id:INTEGER; msg:MESSAGE) is
    local
        remotename:STRING;
    do
        if  <user_id is remote>  then
            remotename := user_table.lookup (user_id);
            remotemailer.bind (remotename); -- bind to a remote instance
            remotemailer.deliver (user_id, msg); -- invocation
        else
            deliver (user_id, msg)
        end
    end

    deliver (user:INTEGER; msg:MESSAGE) is
    do
        ...
    end
    ...  -- other routines
end
```

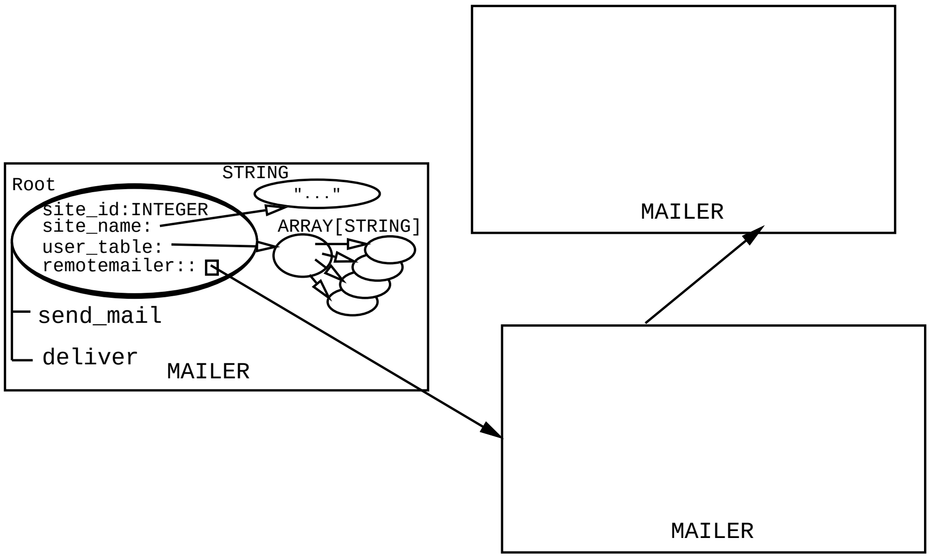

Figure 4: Mailer example: code and representation

distributed system given the ability to start a computation in any cluster that can span across a number of clusters and return results. Another simplistic view would be to look at the model as a set of sequential Eiffel programs executing on different nodes, with each program having the ability to communicate with another program - an actual view would be that of a thread executing one sequential Eiffel program having the ability to enter into and execute in the memory images of other Eiffel programs. Multiple threads can be active in one cluster, and several threads may be executing in the distributed system. With the ability to have multiple threads in one address space (in one object cluster), this programming model subsumes the shared memory programming model, supported by a typical shared memory multiprocessor.

Thus, the Distributed Eiffel language provides a two-tier granularity of objects: large-grained and fine-grained. Large-grained objects are separate address spaces that serve as units of distribution. Fine-grained objects are internal to large-grained objects. The large-grained objects interact with each other by means of *invocations*, while the interaction between fine-grained objects is by local *procedure-calls*. Both invocations and procedure-calls can be *synchronous* or *asynchronous*. In the case of synchronous invocation/procedure-call, the calling thread blocks until completion, while in the case of asynchronous invocation/procedure-call the call proceeds in parallel as a new thread along with the calling thread.

```
hi: PROMISE[INTEGER]; -- typed PROMISE
i: INTEGER;
hl: PROMISE[LINKED_LIST[REAL]];
l: LINKED_LIST[REAL];

routine is
local
    x: EIFFELOBJ;
    y:: CLOUDSOBJ;
do
    ...
    hi:=x!doit (...); -- Async. Procedure call
    ...
    y.bind ("...");
    hl:=y!doit (...); -- Async. Invocation
    ...
    i:=hi.claim; -- claim result (INTEGER)
    l:=hl.claim; -- claim result (LINKED_LIST of REALs)
    ...
end
```

Figure 5: Asynchronous Invocations/Procedure calls and Promises

Invocations (both synchronous and asynchronous) between large-grained objects are directly supported by the Clouds OS. Procedure calls (both synchronous and asynchronous) between fine-grained objects is handled at the language level. During procedure calls between fine-grained objects (which happen within an address space), parameters are passed by reference, except for basic types (like integer, real etc.) which are passed by value. During invocations between large-grained objects, both fine-grained objects and basic types are passed by value, while large-grained objects names are passed by reference (by passing their sys-names).

A *promise* mechanism is used to collect results from asynchronous invocations and procedure-calls (refer to Figure 5).

Support for concurrency control is provided at the fine-grained object level. The scheme is based on the philosophy that each object, not its clients, should be responsible for controlling concurrent activity within it. The passive, shared-space nature of fine-grained objects makes it easy to build synchronization in the routines of each class. The language provides a multiple reader/single writer model of synchronization which can be combined with conditional synchronization. A conditional synchronization scheme allows a thread to wait for some condition to become true. The following syntax is used:

routine {*accesses* | *modifies* | *is*}
when
    <*Boolean_expression*>
do
    ...
end

The programmer can optionally label a routine as a reader (`accesses`) or a writer (`modifies`). He may also provide an optional *when clause* which specifies a boolean expression. A thread invoking the entry point acquires a suitable read or write lock and, if the boolean expression in the *when clause* is true, proceeds to execute the body. If it is not true, it releases the lock and blocks itself until the condition becomes true. When awakened, it tries to re-acquire the lock and evaluates the boolean expression again. A null when clause is assumed to correspond to `true`. Figure 6 shows the definition for the bounded_buffer class. It is assumed that a routine with the `is` tag neither accesses nor modifies the state of the object directly. It can access or modify the state only by calling other routines that have a proper reader/writer tag.

One important advantage of this synchronization scheme is that it allows programmers to fine-tune synchronization ensuring that locking occurs at the lowest granularity possible, thus maximizing concurrency.

For more details refer to [GL92, DCM+90]. The implementation on the Clouds OS has been in operation for more than a year now. Also, a subset of the extensions that allows fine-grained objects to call each other asynchronously has been implemented on the KSR shared memory multiprocessor. Such asynchronous procedure-calls, implemented using the p-threads package on KSR, allows multiple threads to be created on the individual processor-cells of the KSR machine, all executing within a single shared address space.

References

[DCM+90] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The Design and Implementation of the *Clouds* Distributed Operating System. *Usenix Computing Systems*, 3(1), 1990.

[DLA88] P. Dasgupta, R. LeBlanc, and W. Appelbe. The Clouds Distributed Operating System. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[GL92] L. Gunaseelan and Richard J. LeBlanc. Distributed eiffel: A language for programming multi-granular distributed objects. In *Proceedings of the 4th International Conference on Computer Languages*, April 1992.

[Inc] Interactive Software Engineering Incorporated. *Eiffel: The Language, Version 2.3 manual.*

[Mey88] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, 1988.

```
Class BOUNDED_BUFFER[T]
export append, remove, item
feature
    buf: ARRAY[T];
    size, count: INTEGER;
    head, tail: INTEGER;

    Create (sz: INTEGER) is
    do
        size := sz;
        buf.create (size)
    end;
    append (element: T) modifies
        when
            count < size
        do
            head := head mod size + 1;
            buf.put (element, head);
            count := count+1
        end;
    remove: T modifies
        when
            count > 0
        do
            tail := tail mod size + 1;
            Result := buf.item (tail);
            count := count - 1
        end;
    item: T accesses
        -- return, but do not remove, the item
        do
            Result := buf.item (tail mod size + 1)
        end
end
```

Figure 6: A Bounded-buffer class