Distributed ORC (DORC)

Adrian Quark
quark@mail.utexas.edu

May 14, 2008

1 Introduction

ORC is a language designed for orchestrating distributed computations, but it has a significant limitation: all distributed communication must be mediated by site calls, and currently sites cannot be implemented within ORC. This causes problems in two situations:

- In order to implement a trivial distribution task (for example, "open a yes/no dialog box on another computer and get the user's response"), the programmer must implement a site in Java and design a communication protocol to connect to it, which may be a lot of work for such a simple task.
- A large program written in ORC cannot be easily broken into parts and run in a distributed manner. Again, it would be necessary for the programmer to implement sites to run on each distributed system and explicitly handle all the communication between ORC processes.

The goal of this project is to solve these problems by introducing a new syntax for distributed expressions. The programmer simply annotates an existing ORC program to indicate where (on which computer in the distributed system) each sub-expressions should be run, and the ORC interpreter transparently handles all distributed communication. "Remote" sub-expressions use remote computing resources, as does a site call, but otherwise have all the properties of regular ORC expressions, including:

- The same ORC syntax
- The ability to call functions and refer to variables defined in the surrounding ORC program
- The ability to publish multiple values
- The ability to participate in asymmetric composition

2 Usage

DORC introduces one new type of value and one new syntactic construct to the language. The new value type is a "server", which corresponds to a logical computer in a distributed computation[1]. The new syntactic construct is the "remote expression", which specifies that an expression should be evaluated on a specific server.

2.1 Servers and Meta-Servers

A "server" is a logical concept; it is entirely possible to have multiple DORC servers running on the same physical computer.

Every server participates in one and only one distributed computation, and all servers in the computation share the same environment (variables and function definitions). This restriction is inforced by making it impossible to obtain a reference to a server outside of the computation in which it participates. In the context of evaluating a specific expression across two servers, one server (which initiated the computation) is the master and the other server (which is performing the evaluation) is the slave. However in the context of the program as a whole, each server may participate in the evaluation of several expressions simultaneously, and therefore there may be no clear master-slave structure to the overall computation.

To create a new server, a DORC program calls a meta-server, which is simply an ordinary ORC site which returns servers. Each meta-server has a known global name, so that any DORC process may request a new server from it. Typically a meta-server corresponds to a specific physical computer and produces servers which evaluate expressions on that computer, but it is entirely possible to implement a meta-server which acts on behalf of a pool of physical computers, returning servers which may evaluate expressions on any member of the pool.

Currently meta-servers are implemented as standard Java RMI servers, which can be contacted using the built-in site `Remote`:

```
val metaserver = Remote('rmi://address:port/path')
```

Incidentally, the `Remote` site is not specific to DORC, it is just a regular ORC site and can be used to connect to any Java RMI server. The "address:port" portion of the URL specifies where the `rmiregistry` server can be found, and the "path" specifies which object in the registry to contact.

To get a new server from a metaserver, use the method `newServer`:

```
val server = metaserver.newServer()
```

The `server` can be used in distributed expressions, described in the next section.

Use the site `Local` to get a reference to the current DORC server. This is useful if you want to execute an expression on a remote server which executes some inner expression back on the local server.

[1] In distributed computing literature this is usually called a "node", but I have chosen "server" to avoid confusion with the term "node" used in the context of the ORC graph-based implementation.

2.2 Distributed Expressions

A remote expression is of the form `f @ r` where `f` is an arbitrary expression and `r` is a variable which holds a reference to a DORC server. The `@` operator has higher precedence than any other operator. As with site calls, `r` may be an expression instead of a variable, in which case it is equivalent to `f @ x <x< r`.

The meaning of such an expression is: when the value of `r` is available, evaluate the expression `f` on the server specified by the value of `r`. The precise semantics of this expression are discussed in a later section.

2.3 Shared Sites

For the most part, you can think of site calls made in remote expressions as equivalent to the same site calls made in local expressions. For example, if you write a value to a channel in a remote expression, you can read that value from the channel locally.

Of course, if all sites were like this, then distributed programs would be pretty boring because they could never affect the remote server. So many sites are understood to operate relative to whatever server they are called on (the "current server"). As a rule of thumb, any built-in site with a global name affects the current server, while any dynamically-created site affects the server where it was created. Some specific examples:

- Any site which creates a new site allocates it on the current server: `Buffer`, `Cell`, `Ref`, `SyncChannel`, etc.
- Printing sends output to the current server's console: `print`, `println`. If you want to send output to a particular console, use the `Printer` site to create a printer object which will print to the console where it was created.
- `Localhost` and `Local` refer to the current server, because that's useful.

2.4 Examples

Let us consider some examples of distributed programs. These are not very interesting because they are all equivalent to similar non-distributed programs, but they serve to illustrate the variety of communication which may occur between distributed servers. In the following examples, I will assume the existence of sites `c.put` and `c.get`, which put and get to an asynchronous buffer. If the buffer is empty, `c.get` waits to return until a new item is placed in the buffer by `c.put`. I will also assume that a remote server has been created and is available in the variable `r`.

`1 @ r` is the simplest remote expression. It evaluates the constant `1` at the remote server `r` and finally publishes the value `1` back to the local server.

`(1 + 2)@r` is a slightly more complex expression. It actually carries out some computation on `r`, evaluating `1+2` and publishing the result.

`(Rtimer(1) | Rtimer(2))@r` will start two `Rtimers` on `r` and publish values after 1 and 2 time units. This example illustrates that, unlike a site call, a remote expression may publish multiple values.

`Rtimer(1)@r | Rtimer(2)@r` gives the exact same result as `(Rtimer(1) | Rtimer(2))@r`, modulo timing concerns discussed in the next section. This example illustrates that it is possible to use a server multiple times. In this case, the server is used to execute two computations concurrently, but it could also be used to execute multiple computations in sequence, or with any degree of overlap in time.

`c.get() | c.put(1)@r` places a value on the buffer at the remote server, and retrieves it locally. Distributed expressions can communicate via sites just like local expressions.

`(c.put(1) >> let(x))@r <x< Rtimer(1)` evaluates the `Rtimer` locally in parallel with the `c.put(1)` remotely. When the remote server reaches the `let(x)`, it must wait for the local server to publish a value to `x` before it can proceed.

`let(x) <x< (Rtimer(1) | Rtimer(2)@r)` evaluates one `Rtimer` locally and one remotely. The local `Rtimer` publishes a value first, and when it does, all further computation of the parallel remote expression is terminated and no value will be published from it.

3 Semantics

The semantics of a remote expression are closely related to the timing semantics of ORC. Traditionally, ORC sites are classified into immediate sites, whose values are published at precisely-defined times or not at all, and non-immediate sites, whose values may be published after arbitrary delay. `let` and `Rtimer` are examples of immediate sites. The fact that these sites are immediate means that the expression `Rtimer(1) >> 1 | Rtimer(2) >> 2` is guaranteed to publish the values 1 and 2 in that order. This requirement is problematic for a distributed implementation, because starting a distributed expression may involve arbitrary delay.

The simplest solution would be to discard the concept of immediate sites. If all sites are allowed to wait an arbitrary amount of time before publishing a value, any delay introduced by remote communication may be attributed to the delay in some site returning a value. Unfortunately, this means that it is no longer possible for a program to rely on the order that values are published by any expression. Whether this is a problem for typical ORC programs remains a subject for future study.

A slightly more refined solution may be possible. The delay introduced by distributed communication affects the semantics only if it is observable. Therefore, it suffices to ensure that the chain of causal relationships connecting any local event to any locally observable result of evaluation on a remote server (the remote server publishing a value or calling a stateful site) includes some non-immediate site to which the delay in distributed communication can be attributed.

I believe, but have not proven, that the expression `f@r` is exactly equivalent to `LET() >> f >x> LET(x)`, where `LET` is a non-immediate form of `let`, and all stateful sites (such as buffers) are also considered non-immediate. The reasoning behind this is as follows: the local node can only communicate with the remote node via the initiation of the expression, through a stateful site, or through a future. The delay in the initiation of the expression is accounted for by the non-immediate site call `LET()`. Communication through stateful sites is also subject to the non-immediacy of these sites. The publication of a where value cannot by itself convey any timing information, and so can only be given a precise time relative to some event observed via another means.

A good example to illustrate the problems with distributed semantics is:

```
( let(a) <a< (let(x) | let(y)) )@r
    <x< (Rtimer(1) >> 1)
    <y< (Rtimer(2) >> 2)
```

This example shows that even with the `LET`-based semantics described in the previous paragraph, the distributed communication mechanism must guarantee in-order delivery to ensure that the correct value is published by `let(a)`. This problem is subtle enough that a proof of the correctness of the distributed implementation must be provided before the programmer relies on any semantics involving immediate sites.

4 Implementation

I had four goals for DORC:

- Don't hurt non-distributed computation
- Optimize to avoid communication
- Be conservative with optimization
- Keep as much of existing architecture and code as possible

I started from the assumption that an active token must reside on the server which is processing it, then identified related objects which can safely be copied between servers. Such objects include the environment and node graph, which are immutable, but not mutable sites and group cells (futures). Any object which cannot be copied between servers must be shared via a remote reference.

This implies that the following events may require distributed communication:

- Activating a token at the start of a remote expression
- Publishing a value from a remote expression
- Sending arguments to a mutable site
- Returning a value from a mutable site
- Assigning a value to a group cell

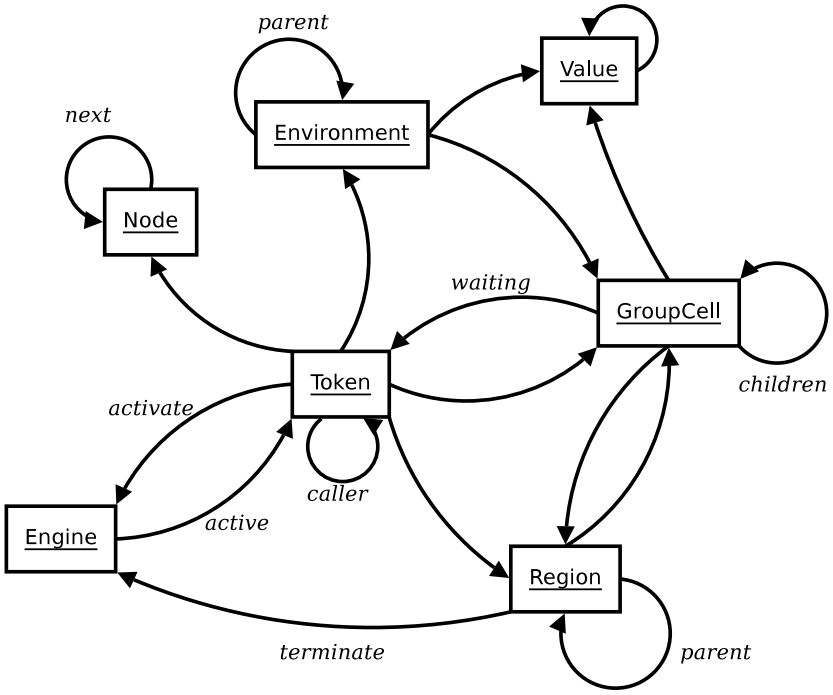

Figure 1: ORC communication

- Notifying a token waiting on a group cell that a value is available
- Killing tokens associated with a group cell which has received a value

Figure 1 shows communication in the non-distributed implementation, where each box represents a class and each arrow represents a reference via which messages can be sent. I have labeled those arrows whose purpose may not be obvious to someone with only a casual familiarity with the implementation.

In contrast, Figure 2 shows communication in the distributed implementation. Solid arrows are local references, while dashed arrows are references to objects which may live on remote servers. You will notice two significant changes, which are discussed in more detail in later sections:

- `GroupCell` has been split into `Group` and `Cell`
- `Site` has been split out of `Value`

4.1 Java RMI

For the actual implementation of distributed communication, I chose Java RMI, which has several benefits:

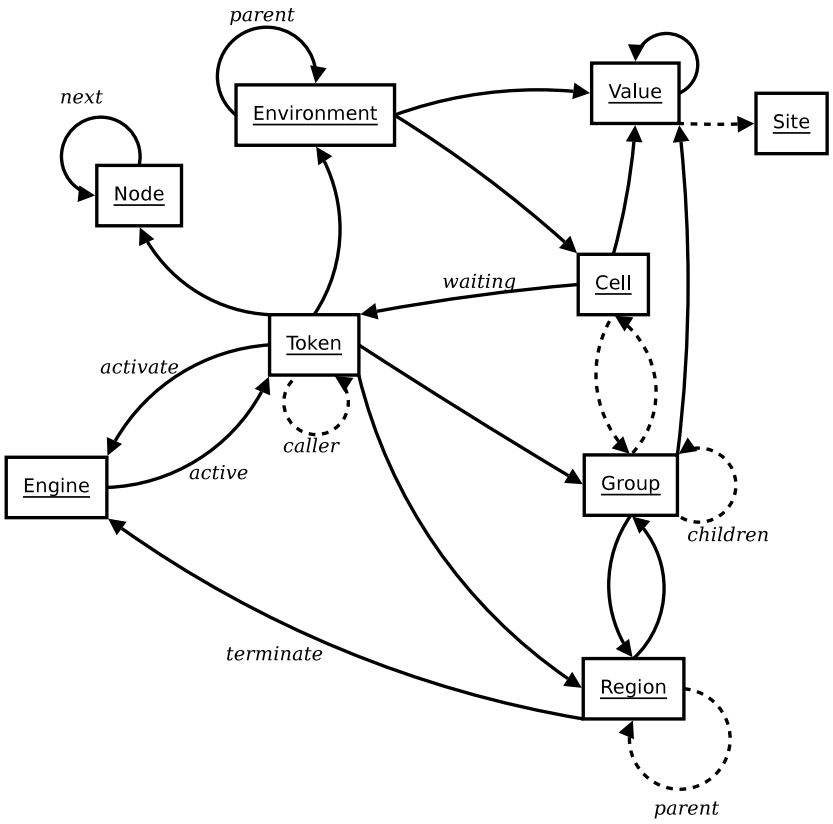

Figure 2: DORC communication

- Objects on remote servers can be called using the same syntax as local objects.
- Arguments to remote methods are automatically serialized. Objects which are registered with RMI as remote objects are serialized to a remote reference which allows remote method calls, while all other objects are copied and not shared between servers.
- The RMI system provides garbage collection for remote references via reference counting. Unfortunately, RMI garbage collection does not collect cycles. It remains to be seen whether this is a problem for typical DORC programs.

Automatic serialization was by far the biggest benefit: the ORC environment may contain arbitrarily-complicated values, and implementing my own serialization routine for such values would be time consuming and also complicate future work on the ORC implementation.

4.2 Remote Expressions

DORC remote expressions are handled exactly like ORC function calls, with only two significant differences. First, since remote expressions are always executed in the environment of the caller, there is no need to switch the environment of the token when it is moved to the body of the remote expression. Second, instead of simply moving the token to the body of the expression, it is serialized together with the expression DAG and sent to the remote server to be evaluated.

4.3 Remote Site Calls

The ORC environment itself is immutable and can safely be copied between servers. The same is true for the majority of primitive values used in ORC programs, including numbers, lists, tuples, and strings. However there are a few mutable sites, such as buffers, which must be shared between servers.

This sharing is implemented by translating references in the environment to such mutable sites into remote references when the environment is copied to a remote server. If the remote server tries to call such a site, it will be a remote site call. When making a remote site call, the arguments to the site and a remote reference to the return token must be serialized and passed to the site via distributed communication. When the site is ready to return a value, it uses the remote reference to send the value back to the token on the local site.

DORC separates sites into an immutable reference value (instance of `orc.runtime.values.Site`) and a site implementation (instance of subclass of `orc.runtime.sites.Site`) which may encapsulate mutable state. Furthermore the interface `PassedByValueSite` is used to mark site implementations which should be copied, either because they are pure or because they should be interpreted relative to the current server.

4.4 Remote Futures

The asymmetric combinator requires careful implementation in the distributed case. The naive approach is to simply allow a group cell to always be handled as a remote reference. This is bad for two reasons.

First, a token must check its group cell every time it is processed in order to ensure the group is still alive. If the group cell is a remote reference this introduces a significant overhead to every token processing step. This can be easily rectified by introducing a local proxy for the remote group cell. The token only has to check the local proxy, which will be automatically notified by the remote group cell when the group is killed.

Second, a token may communicate with a group cell to check if it has received a value, to wait on that value, and finally to be notified when the value arrives. If multiple tokens on the same server all need the value, they will make redundant requests: they will all request the value separately from the group cell, even though some other token on the same server may already know the value. My solution is to introduce a local cache on each server which acts as an intermediary between tokens and group cells.

Instead of every token asking the group cell directly for a value, they ask the local cache. If the local cache does not have the value, it asks the group cell on their behalf. If necessary, the local cache waits for the value and is notified by the remote group cell when a value is ready, so that it in turn may notify all of the tokens waiting on its server.

One important question is whether these optimizations are correct. Specifically, when a group cell is killed, there may be a race condition due to the delay in notifying the tokens working on that group cell. How do we know this delay will not cause problems? Correctness requires that we must guarantee two things:

1. The group publishes exactly one value.
2. It should not be possible to observe progress of tokens in the group after the value is published.

The second requirement deserves further explanation. What does it mean to "observe progress ... after the value is published"? In a semantics with no immediate mutable sites (such as that proposed for DORC), this simply means that there cannot exist a causal relationship between the publishing of a value and some event which occurs on the right-hand side after the value is published. In other words, a token on the right-hand side should not observe any event from the left-hand-side that depends on the value being published, and a token on the left-hand side should not observe any event from the right-hand side that occurs after the value is published.

The following example illustrates both points:

```
let(x) >> d.put(1)
    <x< c.get()
      | (c.put(1) >> d.get())@r
    <c< Buffer()
    <d< Buffer()
```

If requirement (1) is not guaranteed, `x` may receive a value twice. If requirement (2) is not guaranteed, then the remote node may observe that the left-hand side of the asymmetric combinator has made progress by receiving a value via `d`.

Note that with no immediate sites, the following program may legally produce a value:

```
let(x) >> c.get()
    <x< c.get()
      | (c.put(1) >> c.put(2))@r
    <c< Buffer()
    <d< Buffer()
```

The explanation is that even though `c.put(1)` ultimately causes `x` to receive a value, there may be an arbitrary delay between sending the value to the channel and that value being received by the first `c.get()`, during which `c.put(2)` may still legally run. This would only be a problem if the program could observe definitively that `x` had received a value before `c.put(2)` started running, which is covered by requirement (2) above.

In DORC the two requirements are guaranteed easily:

1. is guaranteed by a mutex on the group cell, which ensures only one value may be published at a time and ignores any attempts to publish values after the first has been published

2. is guaranteed by waiting until all tokens in the right-hand side have been notified of group death to proceed with the left-hand side

4.5 Deadlock Freedom

A valuable property of ORC is that any program which does not use mutable sites is free from deadlock. A proof sketch: in the absence of mutable sites, communication between any two concurrent computations is one-way. There can be no communication between the left and right sides of `|`, `>x>` does not enable communication between concurrent instances of the right sub-expression, and `<x<` only allows communication from the right side to the left. Therefore if one side of such a combinator depends on the other for progress, the converse cannot be true, so deadlock is impossible.

DORC does not change the core combinators, so this statement remains true provided the DORC implementation itself is free from deadlock. I have not proven this latter result, but it should be fairly straightforward to do so. Since the DORC implementation uses essentially the same communication structure as the original ORC implementation, assuming the original implementation is free from deadlock, the DORC implementation only has to worry about distributed deadlock. Distributed deadlock can be avoided by ensuring that all distributed messages are non-blocking if they may need to acquire a lock. I accomplish this by running key distributed remote procedure calls in a separate thread, which makes them non-blocking with respect to the main thread.

With arbitrary stateful sites, deadlock freedom is clearly not guaranteed, in either the distributed or non-distributed case. Further research is needed to determine whether there is some useful subset of sites or structures for expressions which preserve deadlock freedom while still enabling more interesting computation.

5 Future Work

Significant future work remains in improving the efficiency of the current implementation. There are three key areas for improvement:

- `LTimer` is broken. It should be possible to fix this using a three-phase commit protocol so that all servers can agree when a logical time unit has elapsed, but implementing this correctly without race conditions may be tricky, especially given that servers may engage in arbitrary communication patterns.

- Remote references are not unpacked when sent back to their originating node. In other words, if a server creates a channel and passes it to another server, which then passes it back, the original server will end up with a remote reference to a local object. Anytime the server uses this object, it will incur needless overhead serializing arguments and transmitting them via the RMI protocol. The solution to this problem involves creating a global identifier for every remote object which

is unchanged as the object is passed between servers. Each local server can keep a cache of identifiers for remote objects it originated, and when it receives such a remote object it can replace it with its local implementation.

- Whenever a remote expression is evaluated, the entire expression is copied to the remote server. Since the expression is immutable, this copying may be unnecessary if the remote expression was evaluated before. The solution is to copy the entire DAG to the remote server when it is used for the first time, and then for subsequent uses it can refer to its local copy of the DAG rather than being sent a new one. One complication is that this requires a global identifier for the node at the start of a remote expression, so that the remote server can be told where to evaluate a token.

- Finally, the entire environment is copied to the remote server whenever a remote expression is evaluated. If the expression only needs a small part of the environment this is very wasteful. Since infrastructure already exists to track free variables, it should be straightforward to identify the free variables in an expression and only copy the portion of the environment they refer to, transitively.

Another important area for future work is in proving the correctness of the distributed implementation, and providing stronger guarantees about timing. My intuition is that this requires a type system which can be used to prove assertions about sites, so that it is possible to automatically verify whether important semantic properties might be violated by distributing an expression.