Implicitly Distributing Pervasively Concurrent Programs

Separating Application Design from Distribution Decisions

[Preliminary report]

John A. Thywissen    Arthur Michener Peters    William R. Cook

2016 Jun 23
Abstract

Distributed programs are often written as a collection of communicating modules. For example, to use Java RMI, programs are divided into objects which can be remotely referenced. Yet, in many cases, it would be desirable to write the program without the program structure being driven by distribution decisions. If distribution is decoupled from program structure, the programming language must allow communication throughout a program’s structure, instead of at a few known points. This situation is simplified if the programming language provides a uniform programming model for local and remote values (location transparency). We present Distributed Orc, which offers location transparency, and distributes program operations automatically in cooperation with the execution environment. By eliminating any special semantics of remote values, Distributed Orc enables programmers to write cohesive distributed programs, rather than programs artificially divided at distribution boundaries. Distributed Orc is derived from the Orc language, a (centralized) concurrent orchestration language.
# Contents

1 Introduction 4

2 Background on Orc 5

3 Introducing Distributed Orc 7

4 Operational Semantics 8
   4.1 Orc Operational Semantics 8
   4.2 Distributed Orc Operational Semantics 10

5 Examples of Distributed Orc 13
   5.1 Web Site Registration Form 13
   5.2 Robot Collaboration 14
   5.3 Data Processing Patterns Imply Distribution 16

6 Related Work 17

7 Future Work and Discussion 18

References 20
1. Introduction

Consider a robot swarm given the task of photographing an object. The swarm needs to decide which members should attempt the task. This decision could be implemented as a centralized program, where finding the closest robots to the object is a simple operation: Scan the list of robots and select robots close to the object. Alternatively, to decentralize this program, one would write code from the point of view of each robot. In this variant, robots would exchange distances to the object with each other and self-select. The centralized variant is simpler and easier to write correctly, since the global behavior is directly specified in the program, rather than derived from the behavior of multiple program instances. When an abstract module is written as a single, integrated concrete module, we call it cohesive. In other words, a cohesive module is one which has not been artificially split to allow distribution.\(^1\)

The system-wide viewpoint of a cohesive program seems to imply that the swarm is controlled by a central controller, or at least an elected leader. This centralization is antithetical to the goals of swarm robotics, and many other distributed systems. Consequently, many distributed systems are written in the decentralized style, despite the resultant complexity. In these cases, the physical architecture constraint of “no central controller” has driven the overall application architecture. This adverse coupling is evident in other domains, such as high throughput applications in data centers. For example, if performance changes require moving work from the Web server tier to the application server tier, a decentralized-style application must be refactored.

Some distributed programming languages have attempted to give developers the ability to write programs cohesively, abstracting away concerns of where each part of the program executes. This language facility is called location transparency. This abstraction has been sharply criticized Waldo et al. (1996), and location transparency is not in widespread use. In current mainstream distributed programming models, such as Java RMI, CORBA, MSRPC, and gRPC, programmers explicitly partition programs into a collection of modules (each executing at a single location) and then explicitly specify each interaction.

The essence of the criticism of location transparency is that semantics of a remote operation should not be “hidden” by appearing as an operation with local semantics. We agree, and propose the reverse: Make the semantics of local operations consistent with the semantics of remote operations.

Two important concerns with remote operations in a distributed system are the possibility of failure and the latency of communication. These concerns are addressed by detecting and responding to failure, and by overlapping operations using concurrency. It may seem excessive to impose these concerns on local operations, instead of just remote operations. However, in modern application architectures, an increasing number of nominally local operations include inter-process communication, which raises these same concerns.

In a conventional language, failure handling and concurrency code can be unwieldy. However, in a language that provides for structured, pervasive failure handling and con-

\(^1\)This concept is inspired by the ideas of cohesion and coupling in object-oriented design, but does not exactly align with that usage.
currency, these issues can be handled deftly. The orchestration language Orc Kitchin et al. (2009); McCord et al. (2013) is such a language.

We have extended Orc into a distributed programming language. Distributed Orc differs from previous work in that it has all of the following properties:

- Local and remote semantics are uniform, allowing consistent asynchrony and failure handling throughout the program (not just at remote calls).
- Communications can occur as part of any expression, not just when crossing object/module boundaries.
- Distributed Orc is general purpose, not restricted to one domain, such as database calls.
- Although the program is written as a single cohesive program, the execution environment does not need a central controller.

This language has been implemented, and can run the examples presented here. This paper: (1) presents Distributed Orc (section 3); (2) provides a formal semantics of Distributed Orc (section 4). (3) shows three programs which demonstrate its utility in multiple domains (section 5); We begin by introducing the Orc language, on which Distributed Orc is based.

2. Background on Orc

The foundation of Distributed Orc is the Orc process calculus and the Orc language which implements it Kitchin et al. (2009); McCord et al. (2013). The Orc language and calculus are designed to facilitate concurrent orchestration, meaning that the Orc program manages various other modules that may or may not be written in Orc. These modules are called sites and they are used to represent everything from arithmetic operations to user interface devices. Sites are called with arguments and publish (return) a stream of zero or more values to the calling Orc program. How these values are computed and the timing of their publication is outside the purview of Orc. This allows enormous flexibility in how Orc programs interact with their environment. Sites are first-class values.

Orc programs are pervasively concurrent; operation ordering constraints are far less common than in conventional languages. The Orc combinators, described below, offer concurrency structures that take the place of conventional control structures. Orc variables are immutable in order to reduce the potential for unwanted data races. When needed, mutable data structures are available through site calls.

Orc extends its immutable variables with transparent futures. A variable can be bound to a future, which is a value that will be available at some later time. When the variable is read, the reader waits for the value (blocks) if it hasn’t been computed yet. This enables concurrent, race-free execution of the writer and reader, but does not require explicit synchronization code.

Orc provides four combinators to process streams of publications from site calls or expressions, e₁ and e₂.
Parallel, written $e_1 | e_2$, executes both $e_1$ and $e_2$ and combines the publication streams into a single stream.

Sequential, written $e_1 >\times> e_2$, provides a way to execute an expression for each publication. It executes $e_1$, and for each publication, executes an instance of $e_2$ with the variable $x$ bound to the publication. Note that $e_2$ is run immediately upon each publication of $e_1$, even while $e_1$ continues to run. The publications of the overall sequential expression are the combined publications of all the $e_2$ instances.

Pruning, written $e_2 <\times< e_1$, provides futures and the ability to pick the first available result. (Note that this combinator is read right-to-left.) Pruning executes both $e_1$ and $e_2$, with the variable $x$ bound to a future in expression $e_2$. The future will be resolved to the first publication of $e_1$, and further execution of $e_1$ will be terminated. If $e_2$ refers to the variable $x$ before it has been resolved, the reference to $x$ will wait for the publication. If $e_1$ halts without publishing, references to $x$ will halt as well. The resulting publications of the overall pruning expression are the publications of $e_2$. The expression $e_2 <\times< e_1$ can also be written as $\text{val } x = e_1$ followed by $e_2$.

Otherwise, written $e_1 ; e_2$, provides failure handling. Otherwise executes expression $e_1$, and then if $e_1$ terminates without publishing, executes $e_2$. If $e_1$ publishes one or more times, $e_2$ is not run. The overall publications of the otherwise expression are either those of $e_1$ or $e_2$.

Orc imposes no ordering constraints on the interleaving of publications from the subexpressions of the parallel or sequential combinators. When the pruning combinator terminates an Orc expression, that expression immediately stops executing within the Orc interpreter. Sites called by the expression will be notified of the termination, but may continue executing since sites are not required to honor the Orc semantics in their execution.

As an example of Orc, consider the following problem: Download a file from the fastest of two servers, and return an error if no server works. In Orc, this is written as:

$$ (d ; \text{error}()) <d (\text{download}(A) | \text{download}(B)) $$

The expression $\text{download}(A) | \text{download}(B)$ executes the downloads concurrently. When a download completes, the pruning combinator binds the result to $d$ and terminates the other download. Finally, the expression $d ; \text{error}()$ publishes an error if $d$ is not resolved to a value. This example shows how Orc simplifies the implementation of orchestration problems which in many systems would require complicated event handling.

The error handling in this example can be extended with timeouts easily:

$$ d <d \text{download}(A) | \text{download}(B) | (\text{Rwait}(10 \times \text{seconds}) \gg \text{error}()) $$

The expression $\text{Rwait}(10 \times \text{seconds}) \gg \text{error}()$ publishes the error after ten seconds, causing all the downloads to abort. This will occur if all of the downloads fail or are delayed for any reason.

Here, we have presented the core Orc calculus. The “surface language” of Orc contains additional constructs, such as if-then-else statements, pattern matching, and list literals, that are “syntactic sugar”. During compilation of an Orc program, these language constructs are translated into core Orc calculus expressions that make use of standard Orc library sites. Full details of the Orc language are available at Orc language project Web...
site https://orc.csres.utexas.edu/, which provides an interactive “Try Orc” Web interface, documentation, examples, and downloadable implementations of Orc.

3. Introducing Distributed Orc

Distributed Orc implements the same programmer-visible language and semantics as Orc, but allows programs to execute across multiple locations. To allow this, Distributed Orc tracks the locations at which data reside and uses this information to determine locations at which to execute program operations. Some data may reside on sets of locations, instead of just a single location.

As a simple example, take \( f() + g() \), where the site \( f \) is available at locations in set \( a \), site \( g \) is available at locations in set \( b \), and site \( + \) is available at every location. Any location in the intersection \( a \cap b \) can execute the whole expression without any need for communication. If \( a \cap b = \emptyset \) or if the current execution is not in \( a \cap b \), then communication is required to execute the expression. For example, \( f() \) could be run on a location in \( a \), and the program could migrate to a location in \( b \), where the \( g() \) and \( + \) operations can be performed.

Distributed Orc builds on the asynchrony and failure-awareness of Orc, and makes additions that can be grouped into three areas:

**Migration** Migrating points of execution in the program among distributed locations.

**Value–Location Relation** Identifying the locations of values in the distributed environment. Distributed Orc tracks both the current locations of a value and the permitted locations of a value.

**Revised Semantics** Extending existing Orc language features to handle remote values, namely: site calls, the pruning combinator, and the otherwise combinator. All other Orc language features operate locally, and therefore are unchanged in Distributed Orc.

Migration of execution in Distributed Orc is driven by the locations of values used in site calls, namely the site and the call arguments. If a site call is to be executed, but the current location does not have local copies of these values, Distributed Orc can copy values between locations (if allowed), migrate execution to another location, or some combination thereof. The locations of existing copies, costs of communication, and any policy constraints on allowed locations of copies may be taken into account by the Distributed Orc implementation when choosing which actions to take.

The pruning and otherwise combinators must be distribution-aware. Execution of a pruning combinator \( e_2 \ll e_1 \) needs to coordinate the selection of the first publication from its righthand expression \( e_1 \) among all locations executing parts of \( e_1 \), and notify other affected points of execution of this selection. Similarly, execution of an otherwise combinator \( e_1 ; e_2 \) must monitor execution, across any involved locations, of its lefthand expression \( e_1 \) for publications and termination.
4. Operational Semantics

Distributed Orc is defined by its formal semantics. The Distributed Orc project uses this formalization in three main ways. First, it is a precise way to define and discuss the language. Second, the implementation follows the communication and location requirements described in the semantics. The examples in section 5 show how the semantics can be used in this way. Third, the formal semantics provide a basis on which to describe extensions (such as migration heuristics) and to prove correctness properties of the language or its extensions.

Distributed Orc’s semantics are formalized using structural operational semantics Plotkin (2004), except for the representation of execution state. Instead of rewriting the program as it executes, threads of execution are represented as tokens which move through the program text, carrying the state of the thread of execution.

4.1. Orc Operational Semantics

Since Distributed Orc uses the non-distributed Orc language as its basis, we first describe the semantics of Orc, which is later extended to Distributed Orc. The rewrite rules for Orc are given in figure 2. The semantics make us of the abstract grammar in figure 1. The notation used for the rules is as follows.

**Token** Tokens enter an expression \( e \), written \( \bullet e \), to begin execution. Result values \( v \) are published by an expression, written \( e \bullet^v \). The notation \( \bullet^v_{\rho, \theta} \) represents a token with environment \( \rho \), tag \( \theta \), and result \( v \).

**Tag** A tag \( \theta \) is a structured identifier for an execution context. Tags encode an execution-stack-like structure associating a token with a particular execution of a combinator or site call. The structure of a tag is its path in the tree of tags.

\[
e ::= c \quad \text{– literal value} \\
| x \quad \text{– variable} \\
| \text{stop} \quad \text{– silent expression} \\
| x(a_n) \quad \text{– site or function call} \\
| e_1 \triangleright e_2 \quad \text{– sequential} \\
| e_1 | e_2 \quad \text{– parallel} \\
| e_2 \triangleleft e_1 \quad \text{– pruning} \\
| e_1 ; e_2 \quad \text{– otherwise}
\]

**Figure 1.** Distributed Orc abstract grammar, as used in semantics rules

---

\(^2\)Previous work on Orc has used a more traditional structural operational semantics, but here we use the same style of semantics we will use for the distributed semantics.
\[ v_{1...n} \neq \diamond \]

\[ \phi \vdash \bullet_{\rho,\theta} x(a_{1...n}) \xrightarrow{s'(v',v_{1...n})} \phi \vdash x \bullet_{\rho,\theta'} (a_{1...n}) \]

where \( \theta' = \text{newChild}(\theta) \)
\( s = \rho(x) \)
\( v_i = \rho(a_i) \)

\[ v \neq \text{stop} \]

\[ \phi \vdash x \bullet_{\rho,\theta} (a_{1...n}) \xrightarrow{\theta'(v)} \phi \vdash x(a_{1...n}) \bullet_{\rho,\theta'} \]

where \( \theta' = \text{parent}(\theta) \)

(SiteCall-Return)

\[ \phi \vdash l \bullet_{\rho,\theta} (l | r) \rightarrow \phi \vdash (l | r) \bullet_{\rho,\theta} \]

(Parallel-Enter)

\[ \phi \vdash (l \bullet_{\rho,\theta}) | r \rightarrow \phi \vdash (l | r) \bullet_{\rho,\theta} \]

(Parallel-PubL)

\[ \phi \vdash l | (r \bullet_{\rho,\theta}) \rightarrow \phi \vdash (l | r) \bullet_{\rho,\theta} \]

(Parallel-PubR)

\[ x \in \rho \quad \rho(x) \neq \diamond \]

\[ \phi \vdash \bullet_{\rho,\theta} c \rightarrow \phi \vdash c \bullet_{\rho,\theta}^c \]

(Literal)

\[ \rho(x) = \diamond_{\theta'} \quad \phi(\theta') = v \]

\[ \phi \vdash \bullet_{\rho,\theta} x \rightarrow \phi \vdash \bullet_{\rho,\theta'} x \]

\[ \phi \vdash \bullet_{\rho,\theta} m(\ldots, x, \ldots) \rightarrow \phi \vdash \bullet_{\rho,\theta'} m(\ldots, x, \ldots) \]

where \( \rho' = \rho[x = v] \)

(SiteCall-Site)

\[ \phi \vdash \bullet_{\rho,\theta} \text{stop} \rightarrow \phi \vdash \text{stop} \]

(Silent)

\[ \phi \vdash \bullet_{\rho,\theta} \text{stop} \rightarrow \phi \vdash \text{stop} \]

(Future)

\[ \phi \vdash \bullet_{\rho,\theta} x \rightarrow \phi \vdash \bullet_{\rho,\theta'} x \]

\[ \phi \vdash \bullet_{\rho,\theta} m(\ldots, x, \ldots) \rightarrow \phi \vdash \bullet_{\rho,\theta'} m(\ldots, x, \ldots) \]

where \( \rho' = \rho[x = v] \)

Auxiliary functions

The semantics use two auxiliary functions to manipulate tags:

- \( \text{newChild}(\theta) \) makes a new child of a tag \( \theta \).
- \( \text{parent}(\theta) \) returns the immediate parent of a tag \( \theta \). 

Figure 2. Orc semantics / Distributed Orc local combinator semantics. For brevity, the rules for pruning and otherwise are only presented in their full form in figure 3, and omitted here.
4. Operational Semantics

**Value** Values are either a concrete value from a site, written \( v \), and taken from the set \( \text{Val} \), or a future with a tag \( \theta \), written \( \diamond \theta \). We write just \( \diamond \) to mean any future. The resolved value of a future with tag \( \theta \) is given by the function \( \phi(\theta) \in \text{Val} \). This function will have no mapping for a tag until that future is resolved. Once resolved, the value is not changed. Futures and the return values of sites are also allowed to have the distinguished value \( \text{stop} \) which signals that there is no value, but the computation has halted.

Figure 2 shows the semantics’ rewrite rules. Site calls wait for any arguments that are futures to be resolved, then notify the environment to execute the call (rule SiteCall-Issue). When the environment returns a result, that result is published (rule SiteCall-Return). If no result is returned the token is halted (SiteCall-Silent). The parallel combinator simply executes both subexpressions and publishes the publications of both (rules Parallel-Enter, Parallel-PubL and Parallel-PubR). The sequential combinator executes the left-hand subexpression (rule Sequential-Enter), and for each publication \( v \), executes the right-hand subexpression with the variable bound to \( v \) (Sequential-PubL). The overall publications of the sequential combinator are those of the right-hand expression executions (Sequential-PubR). We leave exposition of the pruning and otherwise combinators to the Distributed Orc semantics section.

The rules presented here omit Orc’s function call facility, for simplicity of presentation. For our purposes here, function calls can be treated as equivalent to inlining the function body at call sites. However, Orc is capable of general mutual recursion by providing functions with their own closure when they are called. See the Orc references for details.

An Orc program begins execution with a single token at the root of the abstract syntax tree: \( \bullet_{\rho,\theta} e \), where \( \theta \) and \( \rho \) are empty values of their respective types. Tokens that return to the root carry values that are considered publications of the whole program.

4.2. Distributed Orc Operational Semantics

The Distributed Orc semantics modify the Orc semantics of figure 2 with enhanced notation, an enhanced reduces relation, and new rewrite rules. Token notation is extended with a location \( k \) to \( \bullet_{\rho,\theta,k} \).

**Location** Locations are written \( k \) and come from the set of locations \( \text{Loc} \).

**Prototoken** A prototoken represents a “suspended” token which cannot currently execute. The notation \( \odot_{\rho,\theta,k} \) represents a prototoken with environment \( \rho \) and tag \( \theta \) at location \( k \).

**Policy for values** \( \mathcal{P}(v) \in 2^{\text{Loc}} \) is the policy set for value \( v \) — the permitted locations for copies of \( v \).

**Location of values** \( \mathcal{L}(v) \in 2^{\text{Loc}} \) is a set of locations with copies of value \( v \). This is permitted to be an under-approximation, allowing some locations’ view of \( \mathcal{L}(v) \) to be incomplete.

**Location of futures** \( \mathcal{L}(\diamond \theta) \in \{\text{Loc}\} \) is the current location (as a singleton set) of the future \( \diamond \theta \).
4.2. Distributed Orc Operational Semantics

**Location of tokens** $\mathcal{L}(\theta) \in 2^{\text{Loc}}$ is the set of locations with tokens and prototokens that have tag $\theta$. This is permitted to be an over-approximation, allowing some locations’ view of $\mathcal{L}(\theta)$ to be out of date.

We enhance the reduces relation $\rightarrow$ to add the location mapping $\mathcal{L}$. Reductions are now written as $\mathcal{L}, \phi \vdash e \rightarrow \mathcal{L}', \phi' \vdash e'$.

We add two more auxiliary functions: $\text{isLive}(\mathcal{L}, \theta, e)$ is true if and only if there are tokens or prototokens in expression $e$ that are tagged with $\theta$ or a descendant of $\theta$. $\text{eraseTokens}(\mathcal{L}, \theta, e)$ returns the expression $e$ after erasing all tokens and prototokens in $e$ that are tagged with $\theta$ or a descendant of $\theta$. These functions take $\mathcal{L}$ as an argument so that they can determine at what locations to operate, namely $\mathcal{L}(\theta)$.

Distributed Orc’s rewrite rules reuse the Orc rewrite rules in a distributed setting. For every rule in figure 2, every token in the rule must be at the same location $k$. Generally, a local rule of the form:

$$
\Gamma \\
\phi \vdash \Delta \rightarrow \phi' \vdash \Delta'
$$

is an abbreviation for:

$$
\Gamma \\
\mathcal{L}, \phi \vdash \Delta[\bullet^\psi_{p,\theta,\rho^k} := \bullet^\psi_{p,\theta,\rho^k}] \rightarrow \mathcal{L}, \phi' \vdash \Delta'[\bullet^\psi_{p,\theta,\rho^k}, \theta = \bullet^\psi_{p,\theta,\rho^k}]
$$

The rule SiteCall-Issue is explicitly updated in the non-local semantics. Local rules cannot use or update the location information of either tokens or values. Rules that can are called non-local.

When a Distributed Orc token executing in one location needs to continue its execution in another location, the token must be migrated. In the rewrite rules, a token migration from location $k$ to $k'$ is represented as a relabeling of the token $\bullet^\theta_{p,\rho,\psi,k}$ to $\bullet^\theta_{p,\rho,\psi,k'}$.

To call a site, Distributed Orc requires the site and argument values to all be at the token’s current location. Distributed Orc can copy values as needed to any location permitted by the values’ location policies (rule SiteCall-CopyVal). Also, Distributed Orc can migrate the token to another location (rule SiteCall-Migrate). Once the token is at a location that has the requisite local values, the site call is invoked (local rule SiteCall-Issue). The SiteCall-Migrate and SiteCall-CopyVal rules are nondeterministic and permit divergence. The SiteCall-Migrate rule permits migration to any location, which is underconstrained. This is to give the implementation semantic flexibility to execute copying and migration steps as it sees fit. This optimization opportunity is briefly discussed in section 7.

The Orc pruning combinator combines three operations: selection of the first result produced by an expression, the resolving of a future to that result, and the termination of the computation that produced the value. A Distributed Orc expression may have tokens executing in multiple locations. To select the first result produced by this group of tokens, Distributed Orc selects a coordination point when the pruning combinator begins to execute (rule Pruning-Enter). Any result produced by a token executing in the right-hand scope of this pruning combinator is transmitted to the coordination point, and all other tokens in the group’s scope are notified to terminate (rule Pruning-PubR). This termination is represented in the semantics as the application of the eraseTokens function. Since Orc makes no strong ordering guarantees, this weak synchronization is consistent
with the base Orc language semantics. When a future is resolved, rule Pruning-PubR updates the resolution mapping \( \phi \), and rule Future can apply to the expressions waiting
askUser("Pick a username") \(>\) username
if isLegal(username) \&\& isUnique(username) then
  createNewUser(username)
else
  displayError(username)

Figure 4. A user account creation process expressed in Distributed Orc.

for the resolution. Publications from the left-hand expression of a pruning combinator become overall publications of the combinator (rule Pruning-PubL).

The Orc otherwise combinator monitors execution of an expression. If the expression halts without producing a result, an alternative expression is executed. The otherwise combinator is handled very similarly to the pruning combinator: A coordination point is selected and notified of in-scope tokens' results and termination (rule Otherwise-Enter). If the coordination point receives termination notifications from all tokens in scope, and has not received any notifications of results, the alternate expression is run (rules Otherwise-PubL, Otherwise-NoPub, and Otherwise-PubR).

At program startup, the execution environment supplies: (1) the location of the initial “root” token, (2) the policy function \( \mathcal{P} \), and (3) the initial location function \( \mathcal{L} \). The manner of specifying these is not part of the language semantics, and is application-specific.

5. Examples of Distributed Orc

This section shows two example programs that demonstrate the advantages of Distributed Orc. The first example shows how the policy sets of sites cause data migration, and how cohesive programming removes complexity. The second example shows how the dynamic location sets of data can be used to optimize the execution of a Distributed Orc program over distributed data.

5.1. Web Site Registration Form

A Web application is an example of a simple distributed system, with the locations being the user's Web browser and the Web server. Most operations of the Web application must execute in part on the browser and in part on the server. For example, a Web form asking a user to select a username needs to validate the username in multiple ways, some of which must be done on the server. Figure 4 shows how such a form is implemented in Orc. This program is simple and cohesive, but when it is executed in Distributed Orc, location policy cause the execution to be implicitly distributed (using SiteCall-Migrate).

The policy sets for askUser and displayError are \( \{ \text{browser} \} \), and the policy sets of isUnique and createNewUser are \( \{ \text{Web server} \} \). The policy set of isLegal is \( \{ \text{browser, Web server} \} \).

Execution of the user account creation program starts with a call to askUser. Since the askUser site is located at the browser, the execution migrates there and prompts the user to enter a username (using SiteCall-Migrate). The call to isLegal is executed on the browser. If isLegal produces true execution migrates to the server to execute isUnique and
5. Examples of Distributed Orc

// Browser side (JavaScript)
function attemptCreateNewUser(username, callback) { ... RPC call ... }
var username = askUser("Pick a username");
function displayErrorIfNeeded(success) { // Callback
  if(!success) displayError(username)
}
if(isLegal(username))
  attemptCreateNewUser(username, displayErrorIfNeeded);
else
  displayErrorIfNeeded(true);
// Server side (Scala)
def attemptCreateNewUser(username) = {
  if(isUnique(username)) {
    createNewUser(username)
    true
  } else false
}

Figure 5. A user account creation process expressed in JavaScript and Scala.

the logical-and (&&) (using SiteCall-CopyVal to copy all needed values, followed by SiteCall-Migrate). Otherwise, execution continues on the browser, because logical-and is shortcutting and can execute on the browser. Based on the result, either the createNewUser call is made without migrating, or execution calls displayError migrating back to the browser if needed.

To implement the same execution in a typical Web programming environment, the programmer needs to divide the program into two parts and handle the remote calls differently from local calls. Figure 5 shows this pattern using JavaScript on the browser and Scala on the server. This manually implements the same execution and migration pattern as the Distributed Orc implementation produces.

Distributed Orc has simplified the remote calls by providing native support for concurrency. However, local calls in Distributed Orc have not become more complex. Instead, Orc's semantics allows programmers to easily control the concurrency. In addition, Distributed Orc has allowed the programmer to write the program cohesively instead of as two communicating modules.

5.2. Robot Collaboration

In autonomous robotics, all programs need to execute without a reliable controller. However, many collaborations between robots are more easily represented as a single cohesive program. In traditional programming environments, a cohesive program must execute in a single location. Distributed Orc's location transparency and migration support enables us to resolve this conflict. As an example, consider a rover and an Unmanned Aerial Vehicle (UAV) collaborating to determine a safe route for the rover to drive from one point to another. The rover has greater computing power and battery capacity than the UAV, but the UAV may not always be in the communications range of the rover to make use of those resources.
5.3. Data Processing Patterns Imply Distribution

```python
def findRegionsOfInterestOnRoverIfPossible(img):
    rois < rois
    at(rover) >> findRegionsOfInterest(img) |
    Rwait(5 * seconds) >> findRegionsOfInterest(img)

def findRoute(from, to) =
    val img = photographPosition(from)
    val regionData = map(lambda(r) = (r, additionalMeasurements(r)),
                  findRegionsOfInterestOnRoverIfPossible(img))
    val route = planRoute(from, regionData, to)
    if getEndPoint(route) /= to then
        concatenate(route, findRoute(getEndPoint(route), end))
    else
        route
```

**Figure 6.** Collaborative route finding in Distributed Orc

**Figure 6** shows the Distributed Orc implementation of this collaborative route planning problem in terms of a set of high-level primitive sites. Two sites, `photographPosition` and `additionalMeasurements`, take a photograph of a location and take measurements of a region, respectively. These sites use sensors on the UAV, so they must run on the UAV. The `planRoute` site computes a route, given starting and ending points and information about regions in between. The site `planRoute` requires too much memory to run on the UAV, so it is only available on the rover. Finally, the `findRegionsOfInterest` site searches an image for regions that are worth further investigation. It is a fast approximate algorithm that can run on either the UAV or the rover, but it is faster on the rover and consumes enough power to be a significant drain on the UAV’s limited power capacity. So, calls to it should be run on the rover when possible.

The `findRoute` function captures an image of the starting area and other measurements and then computes a route based on the resulting data. Then, `findRoute` recursively calls itself to find the rest of the route to the target if the whole route is not visible in one aerial photograph. The auxiliary function `findRegionsOfInterestOnRoverIfPossible` migrates to the rover and calls `findRegionsOfInterest`. However, if this migration is delayed for more than 5 seconds or fails (for instance, because the rover is out of reliable communication range), the program calls `findRegionsOfInterest` on the UAV. The first result to arrive is used and then the other computation is terminated.

The local and remote operations in **figure 6** are handled consistently within a single cohesive module. This allows the programmer to handle errors consistently as well.

**Figure 7** shows an implementation of the same collaboration in Scala using Scala’s standard concurrency primitives, a high level RPC library, and a distributed flag `TerminationFlag` (to communicate termination requests between locations). Unlike the Distributed Orc implementation, the `findRoute` operation is no longer implemented cohesively. In addition, the remote calls in Scala require future handling while local calls to not allow it, forcing the programmer to manage two mixed programming styles. Finally, manual concurrency and termination handling is needed throughout the program.
5. Examples of Distributed Orc

```
// Rover
def measureForRoute(from, to) = ... RPC call ...
def findRoute(from, to) = {
  measureForRoute(from, to) onSuccessful { regionData =>
    val route = planRoute(from, regionData, to)
    if (getEndPoint(route) != to)
      concatenate(route, findRoute(getEndPoint(route), end))
    else
      route
  }
}
// UAV
def findRegionsOfInterestOnRover(img, terminator) = ... RPC call ...
def findRegionsOfInterestOnRoverIfPossible(img) = {
  val result = Promise()
  val terminator = TerminationFlag()
  val roverCall = findRegionsOfInterestOnRover(img, terminator)
  roverCall.onSuccessful(v => result.success(v))
  timer.schedule(5 * seconds, { () =>
    result.success(findRegionsOfInterest(img, terminator))
  })
  result.future.onSuccessful(_ => terminator.set())
  result.future
}
def measureForRoute(from, to) = {
  val img = photographPosition(from)
  findRegionsOfInterestOnRoverIfPossible(img) onSuccessful { rois =>
    rois map { reg => (reg, additionalMeasurements(reg)) }
  }
}
```

Figure 7. Collaborative route finding in Scala

5.3. Data Processing Patterns Imply Distribution

In Distributed Orc, unmodified functional data processing operations, including map and fold, can execute in a distributed fashion. The function `combineDataAndUpdate` in figure 8 takes a combining function, an initial processing context, and two datasets. It recursively combines the datasets using the function. Each call to the function produces a new context that is passed to the next call (like the accumulator in a fold operation). An example of a combine function would be a log analysis task which needs to compare and merge the events that occurred in multiple logs.

If the dataset storage is distributed across multiple data centers, represented as variations in the location sets of the arguments to `combineDataAndUpdate`, the traversal defined by `combineDataAndUpdate` will migrate from data center to data center as needed (using SiteCall-Migrate). The context will only be copied between data centers (using SiteCall-CopyVal) when migration is required to bring the computation near the new data that it is processing. The Distributed Orc semantics would also allow the whole dataset to be copied to one location, but heuristics could easily determine that the context is cheaper to copy than the dataset.
6. Related Work

Many related approaches to distributed programming have been studied during the past 40+ years. Here, we provide brief coverage of some of the work in the area, but space constraints preclude broader or deeper coverage of the vast body of related work. There are a number of distributed languages with features similar to Distributed Orc.

The Emerald language Black et al. (1987); Jul (1993) was an impressive early leader in this area. Emerald argued for a cohesive programming style, with no explicit communications (RPC, RMI) code. Obliq Cardelli (1995) was a influential language that provided location transparency, but without migration of objects among locations. Distributed Oz Haridi et al. (1997); Van Roy et al. (1997) has strong similarities to the Distributed Orc approach; however, Orc has distinctly different combinators, and does not have the consistency needs implied by Oz’s constraint store.
J-Orchestra Tilevich and Smaragdakis (2002) and JavaSplit Factor et al. (2004, 2006) implement a software distributed shared memory abstraction in the Java VM. These systems, as described, do not attempt to place threads based on any cost model, nor do they support migration.

Dryad Isard et al. (2007) is a dataflow distributed language with a set of combinators that have some similarities to Orc. Unlike Orc, Dryad’s execution engine contains a job manager that maintains a global state of the distributed computation graph to schedule work on cluster nodes.

Demaq/Transcale Böhm and Kanne (2011), at compile time, fragments cohesive programs into per-node sub-programs that then do not need a distributed runtime system. Because this process is at compile time, the distribution of the program must be statically determined, unlike distribution decisions in Distributed Orc. Several languages (Hop Serrano (2007), STIP.JS Philips et al. (2014)) address distribution in the Web development setting using fragmentation.

Cω and Remote Batch Invocation provide a cohesive programming model for certain distributed operations. Cω Bierman et al. (2005), now implemented as Language Integrated Query (LINQ), unifies distributed computation for the case of database queries. Dandelion Rossbach et al. (2013) leverages the LINQ language semantics to target clusters of GPUs and FPGAs. Relatedly, Remote Batch Invocation Ibrahim et al. (2009) handles code blocks in a language which computes on a mix of local and remote objects. Neither Cω nor Remote Batch Invocation support true migration.

The π-calculus has inspired a number of distributed process calculi. Two notable examples are Distributed Join-Calculus Fournet et al. (1996) and Nomadic Pict Sewell et al. (2010). Both provide a restricted form of location transparency and migration for explicitly delineated agents. Migration cannot occur at arbitrary program points.

7. Future Work and Discussion

A Distributed Orc program describes a distributed computation, but does not specify where it should be run, allowing the runtime to make distribution decisions based on data locations. This separates the distribution decisions from the application logic. The semantic rules permit (through their non-determinism) a wide range of static and dynamic optimizations based on the program and its distribution.

For example, static analysis could reduce overhead by determining where parts of the program should run based on where values will be and where the results will be needed. Such an analysis would use the techniques of bi-directional program slicing and distributed dependence graphs Duesterwald et al. (1993) along with a utility function that describes the communication costs of value copying and migration.

As happened with conventional optimizing compilers and JIT runtimes, Distributed Orc optimizing compilers and runtimes will need to explore the design space of optimizations to find effective ones. We believe that these optimizers will make Distributed Orc communication costs competitive with hand-designed communications, but this is an open question at this point.

Alternatively, distribution specifications could be developed for Distributed Orc programs, which partially specify where parts of the program should execute, and how val-
In Distributed Orc, site call failures are handled as in Orc. However, distributed execution introduces the potential for failures that are not possible in Orc, such as loss of executing tokens caused by a node failure. We believe failures at this level should be handled by the Distributed Orc runtime. In some cases, tokens executing on the failed node could be restarted on another node. However, there are a range of semantic and implementation possibilities which will be examined in future work.

The work reported in this paper is a first step in an ongoing program of research. We do not propose that all distributed programming discard current languages in favor of Distributed Orc, but rather we expect languages to evolve facilities similar to those shown here. As programming languages evolve to accommodate the pervasively asynchronous and distributed nature of modern programming environments, languages will develop structured concurrency and failure handling features. These features will make languages amenable to adding Distributed-Orc-like location transparency and distributed execution, and reaping the benefits outlined in this paper.

Acknowledgments.

David Kitchin provided much useful input during the development of this work, particularly with respect to the semantics. David is also responsible for much of the current Orc language and its implementation.

Jayadev Misra originated the Orc process calculus and language. Prof. Misra leads the Orc research group and has provided significant guidance in this work.


Van Roy, Peter; Haridi, Seif; Brand, Per; Smolka, Gert; Mehl, Michael; Scheidhauer, Ralf. Mobile Objects in Distributed Oz. *ACM Trans Programming Languages Syst.* 1997 Sep; **19**(5):804–851. doi:10.1145/265943.265972.