# Cohesive Programming for Distributed Systems

Qualcomm Innovation Fellowship 2014 Proposal

John A. Thywissen
The University of Texas at Austin
jthywiss@cs.utexas.edu

Arthur Peters
The University of Texas at Austin
amp@cs.utexas.edu

## Abstract

Our work seeks to greatly simplify the programming model of cohesive distributed systems. Here, a *cohesive* distributed activity is one that is conceptually an individual operation, yet executes across multiple physical devices. We believe that the communications structure of a distributed program should be separable from the software engineering modularization of the program. Current distributed programming technologies do not allow these concerns to be decoupled. We propose developing a distributed programming model that allows a single program text to be executed across distributed hardware. Module boundaries within the program text need not align with hardware communication boundaries, thus allowing separation of distribution concerns from modularity concerns.

## 1. Distributed Systems are Often Cohesive

Consider the distributed systems aspects of the following robotics system: Multiple robots patrol an area, as a fire watch. The system as a whole must perform many activities including allocating robots to areas, performing navigation, and of course actually reporting fires. These tasks are distributed in a variety of ways, some on board the robot, some to a central controller. Crucially, though, there are some tasks that are performed partly on the central controller and partly on the robots. Circumventing obstacles is one such task: Sometimes the obstacles are other robots, so the central controller, in cooperation with the involved robots, can negotiate a resolution to the conflicting robot paths.

However, each of these distributed controller–robot tasks must behave cohesively, as if it was a single program. For example, the obstacle resolution task requires the fusion of the sensor data indicating an obstacle with the reported positions of the other robots. This sub-task could be decomposed into separate programs which communicate, but it would be superior from a software engineering viewpoint to maintain it as a single program, with various parts of this unified program running on the robots and the central controller.

This example is by no means unique. In many distributed systems, activity that is conceptually an individual operation executes across multiple physical devices. This is true at all scales of complexity, from multi-robot systems to mobile content streaming. Even an operation as unremarkable as user input validation on a Web form is often implemented as a mix of code running in the Web browser and the Web server. Admittedly, many distributed problems naturally decompose into separate communicating modules. However, this module-per-device decomposition is not always the most suitable one.

## 2. Current Methods Shatter Cohesion

Despite the ubiquity of distributed applications, current distributed programming models can be tedious and precarious to use. Programmers are required to express distributed operations as isolated modules, each executing on a single device, and then explicitly and painstakingly specify each interaction. Quickly, these communications details overwhelm the codebase, obscuring the development of the useful application logic.

RPC systems such as XML-RPC and Java RMI do not solve this problem. They still force separation between the client and the server and require large amounts of communication code. Take Java RMI [7], for example: To develop an application that executes on two machines, a developer would create two separate programs—a client program and a server program—which share a Java interface. For the simplest case (a single method call), this adds about 25–50 lines of code overhead. More harmfully, this fractures the application logic into these separate programs. This architecture for RPC is quite common: DCE/RPC, CORBA, ONC RPC, MSRPC, Distributed Ruby, and many other systems impose this client-server split.

Erlang [1] provides a less constrained model of communication. Erlang processes can send messages to arbitrary other processes, and processes can be distributed among nodes. This allows interactions beyond strict client-server style; however, Erlang still structures its code into separate modules among which messages are passed. This results in the same fragmented programming style that RPC often creates.

To be fair, using module boundaries as communications interfaces is a semantically straightforward approach. Interactions across modules are, by design, well-defined and con-

trolled. So, it may seem natural for a language to use its module boundaries to "do double duty". We think this is not the right choice for cohesive tasks that span multiple devices.

## 3. Deployment Decisions Should Not Commandeer Program Structure

*We assert that the communications structure of a distributed program should be a separable concern from the software engineering modularization of the program.* The partitioning of an individual operation across multiple modules gratuitously complicates development, validation, and maintenance of what is actually a simple conceptual operation or application. This assertion is our central thesis that will be developed and evaluated as our proposed project.

Tasks involving allocation, control, and monitoring of multiple agents, such as the multi-robot patrol example, are *coordination* problems. One might expect coordination architectures to accommodate cohesive distributed tasks, but they also suffer from the client-server-style view of distribution. Coordination from a central viewpoint, called *orchestration*, is presented as the alternative to *choreography*, in which the coordination is performed in a peer-to-peer manner among the agents. This coordination dichotomy is a false dichotomy. Wholly centralized orchestration and wholly distributed choreography are two extremes of a continuum, but moving off the extrema calls for novel programming models of distribution.

Some prior work has explored the area of cohesive programming of distributed systems. For instance, C$\omega$ [2], now implemented as Language Integrated Query (LINQ), and Remote Batch Invocation [3] have shown that, for the restricted cases they addressed, it is beneficial to express distributed programs in an integrated manner. Remote Batch Invocation handles code blocks of mixed local and remote computation. Loops and branches in these blocks are "remoted" when they operate on remote data. Dandelion [6], for example, leverages the LINQ-style language semantics to target clusters of GPUs and FPGAs.

Based on these encouraging initial developments, we propose to evolve our already-fully-developed orchestration language, Orc, into a distributed language that realizes our thesis, which we will call Distributed Orc (dOrc).

## 4. Distributed Orc Enables Cohesive Programming of Distributed Programs

Our goal with Distributed Orc is to allow the programmer to write simple operations in simple ways even if they span multiple devices in the physical system.

The foundation of Distributed Orc is the Orc process calculus [4], which provides a structured and principled semantics for concurrent programs, including time sensitivity and failure. The Orc language [5] implements the Orc process calculus, and provides a practical language for implementing concurrent programs. The Orc language and calculus are designed to facilitate concurrent orchestration, meaning that the Orc program manages various other modules or subsystems that may or may not be written in Orc. These modules are called *sites* and they are used to represent everything from arithmetic to a display device to the user. Sites are called with parameters and publish zero or more values back into the program. How these values are computed and when they are returned is outside the purview of Orc. This allows enormous flexibility in how Orc programs interact with the larger world.

As an example of the power of Orc, consider the following problem: From a set of mirrored file servers (call the servers $A$, $B$, and $C$), download a file from the fastest responding mirror, and return an error if no mirror works. In Orc:

$$\textbf{val}\ d = (A \mid B \mid C) >x> \mathsf{download}(x)$$
$$d\ \textbf{;}\ \text{"No mirror responded"}$$

$A \mid B \mid C$ simply returns all server names concurrently, and $\mathsf{download}(x)$ takes a server and attempts the download. $(A \mid B \mid C) >x> \mathsf{download}(x)$ runs all the downloads concurrently. **val** $d =$ expression selects the first result of expression and then terminates expression, so $d$ is bound to the first successful download. Finally, the expression $d$ **;** "No mirror responded" returns the message if $d$ is not bound to a value. So, the code above will return the error if and only if none of the mirrors respond. This example shows how Orc simplifies the implementation of orchestration problems that in many systems would require complicated event handling. Additional Orc examples and an interactive Web interface are available at http://orc.csres.utexas.edu/.

In dOrc, a single program text is annotated to indicate where parts of the program will run, while still maintaining the structured approach of Orc, with its ability to concisely articulate orchestrations. For example, a Web form asking a user to select a username for a new account might have dOrc code similar to:

```
askUser("Pick a username") >username>
if @webBrowser : isLegal(username)
  and @webServer : isUnique(username)
then
  @webServer : createNewUser(username)
else
  displayError(username)
```

where the @webBrowser annotation causes isLegal to be executed in the browser's environment, and the @webServer annotation causes isUnique to be executed on the Web server. askUser and displayError must always run in the Web browser, so they are annotated as part of their declarations.

Given the annotations, dOrc needs to determine the best places to execute the program. The location annotations

specify sets of devices, not a single device. So, the partial order of locations under set inclusion gives a lattice. Standard order-theoretic notions offer possibilities for optimization of execution locations. As a simple example, take $@\mathsf{a} : f() + @\mathsf{b} : g()$. Any device in $a \cap b$ can execute the whole expression without any need for communication. If $a \cap b = \varnothing$, then the expression will have to be executed on two devices. The $+$ operation, since it is not annotated, can be executed on a device that minimizes communication, taking into account the location of its arguments and the destination of its result.

When an expression executes on a different device than its surrounding or "parent" expression, how much of the context needs to be transferred? For example, the expression $\mathsf{isUnique}(username)$ needs the bound value for the name $username$. Transmitting the whole environment with each call isn't acceptable, so some pruning must happen. Part of this is simple to handle—dOrc can prune environments in the same fashion that Orc does when building closures. But, this isn't the whole answer; there is semantic work to be done on this issue.

## 5. Project Plan; First Year Goals

Our research will be guided by the following questions: What source level annotations are practical for writing a single text distributed program? In addition, what exactly are the semantics of these annotations? Once we have defined the dOrc language, we will move on to implementation concerns. What is required of a runtime system to execute such a distributed program over heterogeneous devices? How are site calls handled when they are not available on all devices in the system? These questions will guide our implementation of dOrc.

By the end of the first year, we will complete the theoretic model of Distributed Orc, providing a formal semantics of distribution in Orc; and an implementation that supports distributed programs running across different devices. We will focus on manually annotated distribution, with warnings when the distribution requires very inefficient communication. The implementation will support any platform with a full-featured Java Virtual Machine. As a tangible demonstration of the this first year of work, we will implement the robot patrol example (on the existing hardware of our robotics group). We will implement other demonstrations, such as having a robot acquire and deliver coffee from the Qualcomm Cafe in our building (thus making the cafe an even more important part of our research infrastructure).

Evaluation of dOrc will occur on a variety of platforms. For example, dOrc running in Web browsers (using a JavaScript backend) will allow us to implement the Web form validation example. We intend to make use of the previous Orc on Android work to support distributing parts of a program to mobile phones and tablets.

We expect developing dOrc to be challenging, but the authors have years of experience developing successful languages both at the semantic and implementation levels. In addition, our advisor, Prof. Jayadev Misra, has a distinguished career's worth of experience with distributed computing and formal semantics, including Orc and its predecessors. Our robotics group, led by Prof. Peter Stone, already has the infrastructure to enable our proposed robotics demonstrations. We also are fortunate to have the experience and advice of Prof. Lorenzo Alvizi in distributed systems and Prof. William R. Cook in programming languages available to this project. So, at the University of Texas at Austin, we have the resources at every level, from theory to hardware, to tackle cohesive distributed computing.

## 6. dOrc: Cohesive Distributed Programming

There is a need for better tools for developing distributed applications. dOrc will fill this need by providing a structured and principled approach to concurrency and distribution. Many application areas could benefit from this cohesive approach, from robotics to Web development. In addition, dOrc opens the door to many exciting avenues of further research. Is it possible to automatically detect performance problems in the distributed code? Could the compiler automatically annotate the program? How should failure handling be specified? For instance, if the network fails or partially fails what happens to parts of the program that were communicating?

We are excited about this project; in fact, the more we think about it, the more interesting possibilities we see. We believe dOrc will allow developers to conduct the symphony of modern computing devices.

## References


[1] ARMSTRONG, J. 1996. Erlang – a survey of the language and its industrial applications. In *The Ninth Exhibition and Symposium on Industrial Applications of Prolog (INAP'96)* (Tokyo, Japan, 16–18 Oct 1996). http://www.erlang.se/publications/inap96.ps.

[2] BIERMAN, G., MEIJER, E., AND SCHULTE, W. 2005. The essence of data access in C$\omega$: The power is in the dot! In *ECOOP 2005 – Object-Oriented Programming* (Glasgow, UK, 25–29 Jul 2005), A. P. Black, Ed. Lecture Notes in Computer Science Series, vol. 3586. Springer, 287–311.

[3] IBRAHIM, A., JIAO, Y., TILEVICH, E., AND COOK, W. R. 2009. Remote batch invocation for compositional object services. In *ECOOP 2009 – Object-Oriented Programming* (Genoa, Italy, 6–10 Jul 2009), S. Drossopoulou, Ed. Lecture Notes in Computer Science Series, vol. 5653. Springer, 595–617.

[4] KITCHIN, D., QUARK, A., COOK, W. R., AND MISRA, J. 2009. The Orc programming language. In *Proceedings of FMOODS/FORTE 2009* (Lisbon, Portugal, 9–11 Jun 2009), D. Lee, A. Lopes, and A. Poetzsch-Heffter, Eds. Lecture Notes in Computer Science Series, vol. 5522. Springer, 1–25.

[5] ORC RESEARCH TEAM. 2013. *Orc Reference Manual*. The University of Texas at Austin, Department of Computer Science. http://orc.csres.utexas.edu/.

[6] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J.-P., AND FETTERLY, D. 2013. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, Pennsylvania, 3–6 Nov 2013). ACM, 49–68.

[7] WOLLRATH, A., RIGGS, R., AND WALDO, J. 1996. A distributed object model for the Java system. In *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies (COOTS)* (Toronto, Canada, 17–21 Jun 1996). USENIX, 219—232.