Distributed Framework and Library for C# .NET

Adam Fowles

Abstract— **Distributed computing is the use of distributed systems to solve computational problems. By dividing a problem into tasks which can be completed concurrently by networked computers, distributed systems make computationally challenging problems easier to solve. When writing programs to take advantage of distributed computing resources it can be easy to get bogged down in low level sockets and message passing. To avoid this a framework and library, written in C# have been created with the aim of making it easier to write and understand programs that take advantage of a networked system of computers.**

I. INTRODUCTION

As computing systems get larger the scale and complexity of problems we wish to solve grows accordingly. Often it is easier to find raw computing power than it is to understand how to split up a computationally challenging problem into more manageable tasks. Large scale enterprise systems exist that can manage warehouses full of hardware efficiently and effectively. These systems handle essential distributed computing challenges such as fault tolerance, redundancy and scalability.

Where they are less effective is in teaching and explaining how to solve common types of problems that are well suited for a distributed or parallel environment. To differentiate terms, parallel computing generally deals with processing units that have some type of shared memory. Processors are then able to communicate by exchanging information utilizing these shared memory resources. Distributed computing on the other hand deals with processors that have their own memory and must communicate via message passing across a network. This distinguishes a distributed system from a single machine with many cores. While the two ideas of parallel and distributed computing are not mutually exclusive for the most part this project will focus on the distributed side of things.

To teach distributed computing techniques in a way that students can relate to a system needs to be in place that takes away some of the challenging aspects of building the underlying system while being flexible enough to allow them to write code that is not tightly coupled with any architecture dependent aspects such as number of nodes. A student should not need to worry about communicating with an underlying system itself but should only need to following the API specifications to run a program.

To accomplish the task of creating a system that helps enforce common principles of distributed computing in an academic setting a library written in C# .NET has been created. Why .NET? With the recent addition of .NET core to the .NET platform, cross-platform code can now be written [1]. While this is not new to languages such Java, frameworks have previously been written to accomplish a similar task; see PJ2 [2]. What makes .NET core different is that it supports multiple languages and paradigms. C#, an object oriented event driven language, was choose for the implementation of the framework and library, however any language within the .NET core platform can be used to write code and it will compile and run. Currently those languages are limited to C# and F# with visual basic coming soon but even the addition of F#, a more functional language, adds flexibility to the developer and is more interesting to work with from an academic point of view.

The idea behind the separating the framework from the library is to abstract away the low level details that go into building a distributed system such as socket manipulation, message protocols and communication between nodes. While important concepts in their own right, they distract developers from concepts involving the structuring of programs to complete computational challenging problems using a distributed system. The framework consists of code used to setup backend nodes, communicate between hosts and transmit data such as files and other message protocols. This code is hidden and inaccessible to users of the DLL, unless compiled differently by the user themselves. The library contains classes and code to facilitate developing a distributed program. This includes classes to set up jobs with tasks and change the behavior of how the backend nodes break up and complete the work.

II. BACKGROUND

A. Running with .NET Core

The .NET core functions differently than other standard .NET framework projects and code. Setting up and running a project has different implications. A substantial difference right off the bat is the use of a JSON project settings file. This file contains all the dependencies needed to compile and run your code. Behind the scenes .NET core uses NuGet as a package manager to install different dependencies. Pip in python functions in a similar fashion as do other Linux package managers like yum. For example if your project needs ”System.Threading.Thread”, you would specify that in dependencies section and when run .NET will make sure that package is available.

The program itself that is used to compile and run code, aptly named, is ”dotnet”. dotnet builds and then runs an instance of your program in the background on the target system. This tool is command-line based and in general the system being developed will deal solely with command-line applications. The .NET core does support other models such as ASP.NET but they are outside the scope of this

project. Two terms to understand when deploying .NET applications are Framework-dependent deployments (FDD) and Self-contained deployments (SCD).

Framework-dependent deployments only involve your code along with any third party libraries. They use the version of the .NET core that is present on the target machine. This is useful if you want a small deployment that does not need to worry about the operating system that it is being deployed too. The downside of this approach is that you need the correct version of the .NET core installed for your application to work correctly. This means if you build an application for .NET version 1.0 that version needs to be installed properly on the target machine.

Self-contained deployments contain everything mentioned above and also the .NET core version your application is running with. SCD need to specify all the target operating systems that will be used to run the application. This is a good approach if the target system does not have .NET core installed however these deployments tend to be on the larger side.

For this project a framework-dependent approach was used since, working in an academic setting, we have the advantage of setting up the proper .NET core installation all target machines. As newer versions of the .NET core are released they can be easily installed and the library can be recompiled against them to support a wider number of features.

B. Parallel Code

While distributed systems are a useful tool for completing tasks often times we want to build upon that by also writing parallel code. The reason that the framework and library are not distributed *and* parallel is that C# provides a compressive, easily accessible parallel library as part of the .NET framework which can be used in the .NET core framework as well. This library is called the Task Parallel Library. It is based on the idea of a task or unit of work. A task can be compared to a thread and may be run as such behind the scenes. This idea of splitting larger tasks or jobs up into a more manageable size occurs on the distributed side of things as well. How those tasks are broken up is dependent upon the type of problem being worked on however there are common solutions that are used frequently when load balancing.

III. Related Work

IV.

References

[1] .NET Core `https://docs.microsoft.com/en-us/dotnet/articles/core/index`

[2] Alan Kaminsky, BIG CPU, BIG DATA: Solving the World's Toughest Computational Problems with Parallel Computing. CreateSpace, 2016.