Table of Contents

[1. MINGLE 1](#_Toc400725797)

[2. Cluster Architecture 2](#_Toc400725798)

[2.1. Cluster Shared Storage 2](#_Toc400725799)

[2.2. Orchestrator (OrchestratorService.exe) 2](#_Toc400725800)

[2.2.1. CheckInService 2](#_Toc400725801)

[2.2.2. OrchestratorService 3](#_Toc400725802)

[2.3. Tracker 4](#_Toc400725803)

[2.3.1. Tracker auto-discovery and Check-in Interval 4](#_Toc400725804)

[2.3.2. Job requests and execution 4](#_Toc400725805)

[2.3.3. Job returns 4](#_Toc400725806)

[2.4. MINGLE Clients 4](#_Toc400725807)

[2.4.1. Implementing Your Own Client 4](#_Toc400725808)

[3. Cluster Configuration 5](#_Toc400725809)

[3.1. Configuration Files 5](#_Toc400725810)

[3.1.1. Tracker Configuration File 5](#_Toc400725811)

[3.1.2. App.config Files 5](#_Toc400725812)

[3.2. Example Setup 6](#_Toc400725813)

[4. Cluster Security 6](#_Toc400725814)

[Appendix A. Sample Tracker Configuration File 7](#_Toc400725815)

[Appendix B. Sample Orchestrator App.config 8](#_Toc400725816)

[Appendix C. Sample Tracker App.config 9](#_Toc400725817)

# MINGLE

MINGLE aims to increase model throughput by orchestrating the execution of multiple Envision jobs on separate physical machines.

An example use case: *automatically run several Envision scenarios in parallel.* An Envision model might define several scenarios that the model should be evaluated under. Since these scenarios can always be run independently of each other, it makes sense to run them in parallel, but running them on a typical desktop workstation would exceed available resources. The framework solves the problem by allowing a user to specify which scenarios to run when a model job is submitted. Each scenario is treated as an independent job and the framework farms the executions to a cluster of compute nodes.

MINGLE has been tested and seems to run pretty well on a cluster of eight Windows 7 virtual machines with 4GB ram and single CPU core.

# Cluster Architecture

The MINGLE framework has a three-tiered architecture: An O*rchestrator* acts as a middle tier that monitors and oversees cluster operation and also services all client requests (Section 2.2). *Trackers* fork and exec Envision processes and then monitor the process until it exits (Section 2.3). Finally, *MINGLE Clients* connect to the orchestrator to submit jobs and request cluster status (Section 2.4).

There is also some shared storage that provides a way to distribute Envision project data to the Trackers (Section 2.1).

## Cluster Shared Storage

The cluster needs shared storage that is accessible by the tracker nodes. The shared storage houses Envision project folders that the trackers will copy to their local file system and then copy Envision’s “Outputs” directory back to. The share needs to be a regular old Windows shared folder.

The path to this shared directory is specified as a UNC path in the tracker configuration file (Appendix A). The Envision project and the data should be totally self-contained in a single folder. This storage needs to be a Windows share accessible from the tracker nodes. When a tracker receives a job from the orchestrator, the tracker copies the entire project folder into local storage and points an Envision process at the project file (the .envx file contained within the folder).

## Orchestrator (OrchestratorService.exe)

The OrchestratorService.exe program actually exposes two services:

* CheckInService: Consumed by Trackers to notify the Orchestrator of status information and tracker job requests (Section 2.2.1).
* OrchestratorService: Consumed by MINGLE clients to push jobs and request cluster status info (Section 2.2.2).

**Note**: Both of these services have poorly chosen names. A better name for CheckInService would have been TrackerCheckInService and OrchestratorService might have been called MingleClientService for clarity.

Exposing both services makes the Orchestrator node a middle man between MINGLE clients and trackers. Trackers push status updates and request queued jobs from the orchestrator. In turn, clients request tracker stats from the Orchestrator and submit new jobs to be scheduled. The orchestrator queues these new jobs for execution and returns job descriptions to trackers when they request a new job. When a tracker checks in, it sends status information and the orchestrator simply copies the status information into its local copy. This local copy is what is queried by the clients when they request cluster status information so the data may be a second or two stale.

### CheckInService

The tracker facing CheckInService is exposed in ICheckInService interface. It defines three operations that trackers can perform. See Figure 1 for the code listing.

Check-ins are done periodically the entire time a Tracker is running. Each call to CheckIn() delivers a TrackerData object that encapsulates status information about the tracker.

Job requests are done only when a job is not currently running. The JobData object contains everything the Tracker needs to know about running an Envision job, including the location of the Envision project folder to copy. The project location specified in the JobData is expected to be relative to the base directory specified in the Tracker’s configuration file (Section 3.1.1 and Appendix A).

public interface ICheckInService

    {

        // Allow Trackers to submit status updates

        [OperationContract]

        bool CheckIn(TrackerData td);

        // Trackers request new Job descriptions.

        [OperationContract]

        JobData RequestJob();

        // Trackers return job descriptions when done.

        [OperationContract]

        void ReturnFinishedJob(JobData j);

    }

Figure : The ICheckInService is exposed to the Trackers. TrackerData objects encapsulate tracker status information. JobData objects encapsulate descriptions of Envision simulations that should be queued to run, are running or have been run.

### OrchestratorService

The client-facing OrchestratorService is exposed via IOrchestratorService and services mingle clients with Tracker status information and information on what jobs are running.

public interface IOrchestratorService

    {

        [OperationContract]

        bool QueueJob(JobData job);

        [OperationContract]

        bool RemoveJob(string jobGuid);

        [OperationContract]

        TrackerData[] TrackerStatus();

        [OperationContract]

        JobData[] AllJobs();

        [OperationContract]

        bool Ping();

        [OperationContract]

        int NumWaitingJobs();

        [OperationContract]

        int NumRunningTrackers();

        [OperationContract]

        int NumIdleTrackers();

    }

Figure : The mingle clients consume the service exposed by IOrchestratorService. Currently, it only allows clients to query cluster status and submit new jobs.

## Tracker (TrackProcessClient.exe)

The tracker monitors envision processes and returns simulation results when the job finishes. It also tries to gracefully handle an instance when Envision crashes unexpectedly. If the tracked process generates a non-zero return value the job is returned as a failed job.

### Tracker auto-discovery and Check-in Interval

When a tracker node is brought online it broadcasts its presence via multicast. If an orchestrator node is present to receive the broadcast, it will respond with its endpoint address. The tracker notes the orchestrator’s endpoint and begins a periodic check-in cycle with the orchestrator. During a check-in the tracker transmits status information as a serialized TrackerData struct.

Currently, the check-in intervals are hardcoded in TrackProcessClient.cs as a number of milliseconds. One could add these options to an instance of the ConfigParser to add flexibility.

private const int CHECKIN\_INTERVAL\_NORMAL = 1000;

private const int CHECKIN\_INTERVAL\_SLOW = 5000;

Normally, the trackers check in at the interval specified by CHECKIN\_INTERVAL\_NORMAL. If the check-in times out, then the interval is increased to the value of CHECKIN\_INTERVAL\_SLOW until the orchestrator responds again. Future work could include increasing the check-in time if failures persist.

### Job requests and execution

When a tracker is IDLE it asks the orchestrator for a new job every CHECKIN\_INTERVAL\_NORMAL milliseconds. The job requests are performed in addition to the check-ins. When a new job is received, job requests are halted and the job is executed.

### Job returns

Once the tracker is finished with a job, the results are copied to a results directory--specified relative to the shared data location--in the tracker’s configuration file (Section 3.1.1). After the simulation results are copied to the shared storage location, the JobData for that job is returned to the orchestrator. Finally, the tracker’s job request timer is started again.

If the orchestrator goes down during job execution and the JobData cannot be returned, the JobData is queued and returned when the orchestrator comes back online.

**Note:** queue states are not saved! So, if there are jobs waiting to be returned and a tracker goes down, that data is lost. Rest assured, this does not mean simulation results are necessarily lost. Data transfer between the tracker’s local file system and the shared storage is independent of the orchestrator.

## MINGLE Clients

### Implementing Your Own Client

If you implement your own MINGLE client, you can use the classes in the EnvwsLib library to help connect and communicate with the Orchestrator. The class OrchestratorServiceClientProxy implements the IOrchestratorService interface. Just create an OrchestratorServiceClientProxy instance and begin calling the methods in it. The class is located in EnvwsLib\Client.

# Cluster Configuration

The Orchestrator needs to be started before the tracker nodes. If the trackers can’t find the orchestrator when they first start they will exit after a brief timeout (a script to restart the trackers if they exit will keep things running smoothly).

It would be ideal to have the tracker nodes start TrackProcessService.exe when Windows boots, but be sure that the user which starts the executable can also run Envision and has read/write access to the shared storage.

**Note:** Since TrackProcessService.exe forks and execs Envision, TrackProcessService.exe should be started so that a UI can be popped up. I never spent the time to figure out how Windows handles running a program as a user without an active session (or access to the window manager).

## Setting up the Operating System

A few modifications to Windows 7 need to be made.

### Disabling the “Searching for a Solution” dialog

If the executable that TrackProcessService is running crashes unexpectedly, then windows might show that *annoying* “Searching for a Solution” dialog. The process does not die until the dialog is closed. This causes trouble because the tracker will think the process is still running and the job will never complete. Worse, closing the dialog requires user interaction. The easiest way to get around this is to disable the dialog for the entire system by editing the registry. Figure 3 shows a PowerShell command to do this.

# create reg. key for turning off the "Searching For a Solution" dialog box.

New-itemproperty -path hklm:\software\microsoft\windows\'windows error reporting' -name DontShowUI -propertytype DWORD -Value 1

Figure : PowerShell command to disable Windows Error Reporting.

Also, take advantage of any quiet/silent modes that your tracked process may have.

## Configuration Files

### Tracker Configuration File

See an example configuration file in Appendix A. The example is commented well enough that no explanation is necessary here.

### App.config Files

Windows WCF programs can be configured with those funny App.config files common in .NET. They are basically an XML file which specify service endpoints. It seems that every endpoint needs one. So the service providers (Orchestrator in this case) and service clients (Clients and Trackers for us) each have one. If you end up writing a MINGLE Client, I would try to get away without using one.

An endpoint is just the familiar <http://URL:port/service> that a service is reachable at. An Example for the orchestrator check-in service that is running on the same host as a tracker would be: <http://localhost:8001/checkin>. This would only work if the tracker and orchestrator were both located on the same machine.

**Note:** For the Orchestrator, you need to specify two end points in the App.config. One endpoint is given to the Tracker’s when they first broadcast their presence to the Orchestrator. Subsequent tracker-to-orchestrator connections are initiated with this endpoint. The second endpoint is used by the MINGLE clients. The clients connect to the same URL as the trackers, but on a different port (because they are connecting to a different service, explained later in this document).

Appendix B shows the orchestrator App.config included in MINGLE.

The trackers are auto-configured, so just use the supplied App.config file, no modification is necessary. You may not actually need an App.config for them, I have not experimented with it.

## Example Hardware Setup

The BSU cluster consists of eight Windows 7 virtual machines with 4GB RAM and a single CPU core per machine. Of the eight machines, one machine functions as a sort of frontend (FE) node (however, every node is accessible from outside the firewall). The FE node also runs the orchestrator and provides the shared storage the trackers use to obtain project data from and copy simulation results to. The entire cluster is behind a firewall that only allows RDC connections from the outside. Every node is accessible via RDC from user desktops. I primarily did cluster admin by RDC-ing into the FE node and then executing remote PowerShell commands to the seven other nodes.

## Example Virtual Machine Setup

For testing I used this configuration for a windows virtual machine base image.

|  |  |
| --- | --- |
| OS | Win 7 Enterprise Evaluation 90-days |
| Hostname | ENVWS-BASE |
| Network | Bridged, dhcp |
| Powershell | V4.0  Download File: [Windows6.1-KB2819745-x64-MultiPkg.msu](http://download.microsoft.com/download/3/D/6/3D61D262-8549-4769-A660-230B67E15B25/Windows6.1-KB2819745-x64-MultiPkg.msu)  Download Page: [Link](http://www.microsoft.com/en-us/download/details.aspx?id=40855) |
| .NET | 4.5 |
| VC Redist | for VS2013 (12.02.xxxx)  Download File: [vcredist\_x64.exe](http://download.microsoft.com/download/2/E/6/2E61CFA4-993B-4DD4-91DA-3737CD5CD6E3/vcredist_x64.exe)  Download Page: [Link](http://www.microsoft.com/en-us/download/details.aspx?id=40784) |
| Chocolatey | [Link](https://chocolatey.org/) |
| Boxstarter | v 2.4.15  [Link](http://boxstarter.org/) |
| Remote tools for  debugging | Instructions: <http://msdn.microsoft.com/en-us/library/bt727f1t.aspx>  Download: <http://go.microsoft.com/fwlink/?LinkID=386600> |

# Cluster Security

Communications between all components is not secure. Make sure your cluster is deployed on a secure network.

1. Sample Tracker Configuration File

; Service configuration options

;

; Working directory where files are downloaded to.

; This should be local to the tracker.

BaseDirectory = D:\ews\_working

; Absolute path to the envision executable.

EnvExePath = C:\envision\envision.exe

; The name of the folder that Envision writes simulation results to.

; This is a subdirectory of the project directory.

EnvisionOutputDirectoryName = Outputs

; The name of the Envision log file that will be copied from the

; Envision directory into the logs folder

EnvLog = Envision.log

; The path to the shared storage that projects may be fetched from and

; results may be copied to.

RemoteBaseDirectory = \\shared\storage\

; A subdirectory of RemoteBaseDirectory where

; envision log files from each scenario are copied into

ResultsLogDirectory = logs

; A subdirectory under RemoteBaseDirectory that the tracker will copy

; simulation results to. This is the directory that users will retrieve

; their results from.

ResultsDirectory = output

Figure 4 The example tracker configuration file. See tracker.conf in the source for this template.

1. Sample Orchestrator App.config

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

    <system.serviceModel>

        <bindings />

        <client />

        <services>

            <service name="EnvwsOrchestrator.OrchestratorService"

                     behaviorConfiguration="OrchestratorServiceBehavior">

                <host>

                    <baseAddresses>

                        <add baseAddress="http://***<orchestrator-uri>***:8001/orchestrator"/>

                    </baseAddresses>

                </host>

                <endpoint address=""

                          binding="basicHttpBinding"

                          contract="EnvwsLib.ServiceContracts.IOrchestratorService">

                    <identity>

                        <dns value="localhost" />

                    </identity>

                </endpoint>

                <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>

            </service>

            <service name="EnvwsOrchestrator.CheckInService" behaviorConfiguration="CheckInServiceBehavior">

                <host>

                    <baseAddresses>

                        <add baseAddress="http://***<orchestrator-uri>***:8002/checkin"/>

                    </baseAddresses>

                </host>

                <endpoint address=""

                        binding="basicHttpBinding"

                        contract="EnvwsLib.ServiceContracts.ICheckInService">

                    <identity>

                        <dns value="localhost" />

                    </identity>

                </endpoint>

                <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>

            </service>

        </services>

        <behaviors>

            <serviceBehaviors>

                <behavior name="OrchestratorServiceBehavior">

                    <serviceMetadata httpGetEnabled="True"/>

                    <serviceDebug includeExceptionDetailInFaults="True"/>

                </behavior>

                <behavior name="CheckInServiceBehavior">

                    <serviceMetadata httpGetEnabled="True"/>

                    <serviceDebug includeExceptionDetailInFaults="True"/>

                </behavior>

            </serviceBehaviors>

        </behaviors>

    </system.serviceModel>

    <startup>

        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />

    </startup>

</configuration>

1. Sample Tracker App.config

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

    <system.serviceModel>

        <bindings />

        <client />

    </system.serviceModel>

    <startup>

        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />

    </startup>

</configuration>