### Implementation of the CCRD Use Case

Our CCRD use case includes:

* Networking between nodes, being able to send arbitrary messages back and forth
* Security/encryption/authentication between nodes
* Entries can be committed into the distributed log
* Cluster can maintain service upon node failure

To produce a functioning prototype for this project it’s necessary to not only write a UDP socket level networking library (with security), but also to implement the complete Raft Algorithm described in [the paper](https://raft.github.io/raft.pdf)**.** It’s very atomic in the way that all the networking and Raft Consensus functionality needed to be implemented to simply fulfil the CCRD, or the prototype simply couldn’t fulfil the CCRD. This means that the prototype is able to stand on its own as a completed product and may start to be used by its intended audience, as the milestone of this subject indicates the case should be. Going forward into next session, the goal is to add more desired features and optimize the library.

To display the library being able to implement the CCRD, we’ve recorded a video of us stepping through the CCRD on a local computer. As will be discussed in the next session, we’ve actually implemented our library into a prototype WinForms application as a UAC/UAS combo to demonstrate functionality. This prototype is developed entirely separately and uses the code libraries produced by this project through the publicly accessible NuGet package. This is a functioning example of the prototype being executed in the production environment.

To create the nodes quickly and locally we’re using a bootstrapping utility. It simply sets the configuration options for each of the consensus nodes, then executes them. These nodes are executed as separate .exe applications on the same local computer using localhost (127.0.0.1) IP/port networking to communicate to one another. Practically multiple nodes won’t be run on the same computer, this is just a demo of it working and communicating using networking ([online video](https://youtu.be/Gx3us19UYaw)).

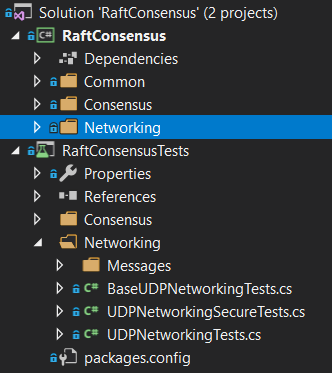


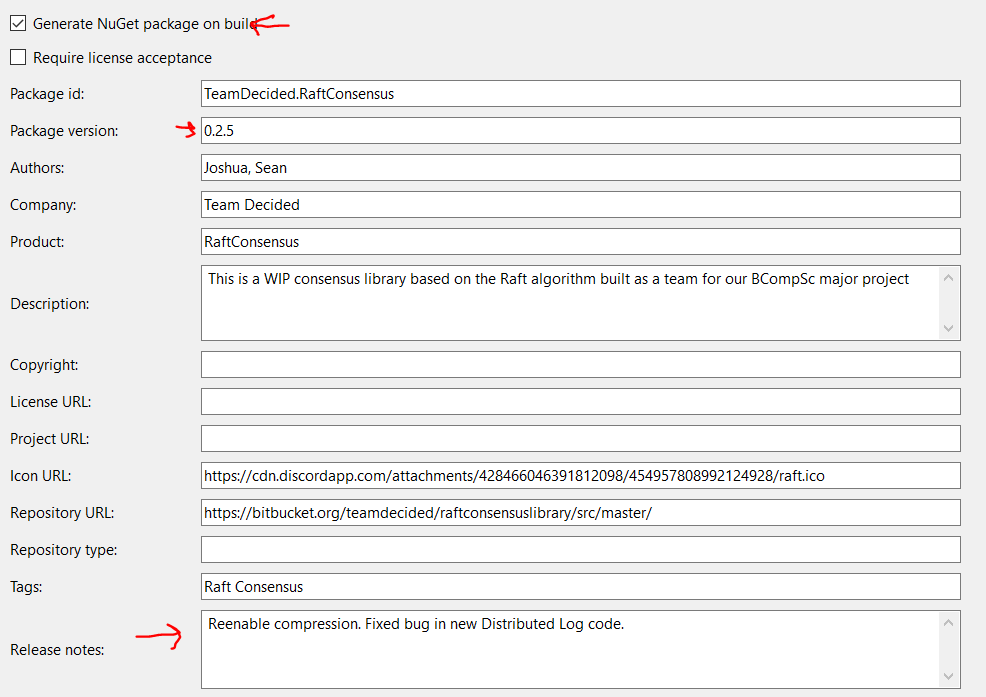
For demonstrative purposes we’ve also created an installer for our prototype demo, [available here](https://cdn.discordapp.com/attachments/428466046391812098/457756399943024650/RaftPrototypeInstaller.msi). Simply install, open the shortcut on the desktop or start menu, and it’s possible to follow along the steps in the video. You can then uninstall as normal in Windows “Programs and Features”.

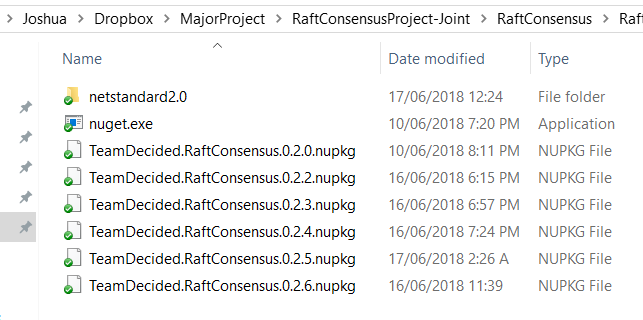
However, for a better display of production environment, we’ve also recorded a video of us stepping through the CCRD running across the Internet using pre-made configuration files to start each node. This is showing that the consensus algorithm not only works locally on the same computer for demonstrative purposes, but also across the Internet as is its literal intended production environment ([online video](https://youtu.be/fekhJKg5pXk)).



## EA Executing in Production Environment

We write the consensus library in a solution in Visual Studio, including all unit testing (Image 6.3.1).

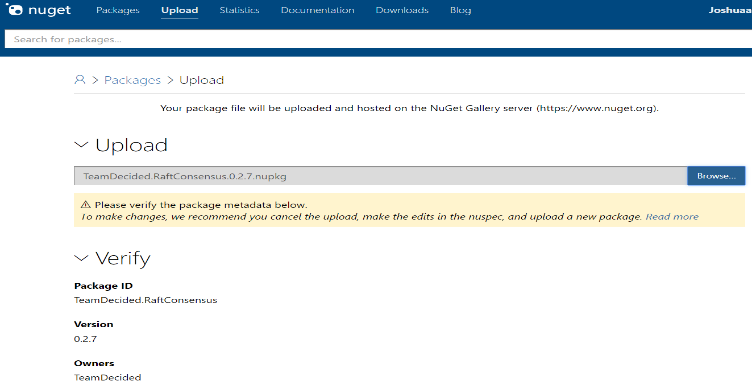
On a Release build, we increment the version number, and write release notes {image}.

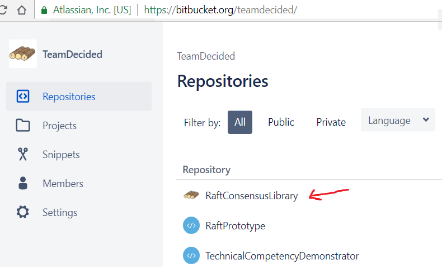


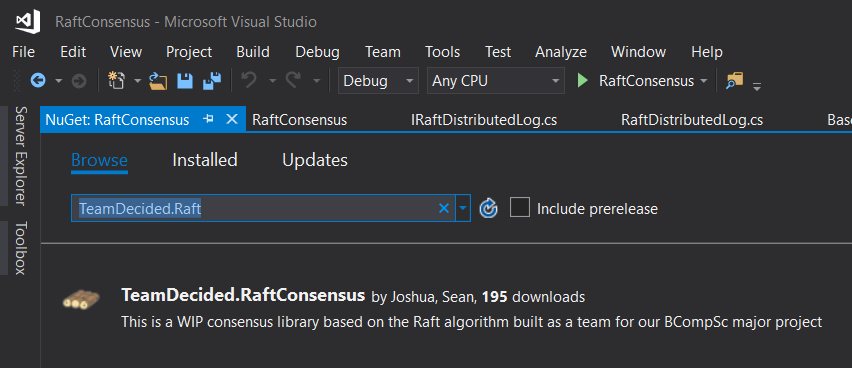
This build creates an NuGet file {image} which we push up onto the online Package Manager.

Then after about 10-15 minutes the package is available publicly for people to download and implement into their software. {image}.

We commit our code to our own Bitbucket repo {image}.





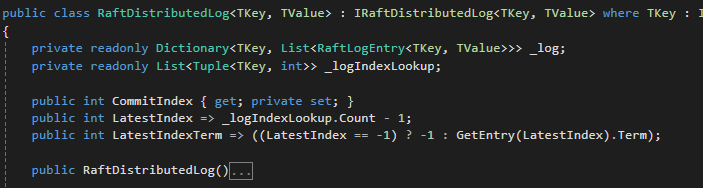
Our production environment consists of a separate developer being able to pull down our code library from the online package manager NuGet {image}, implement it into their program, and turn their UAC into a fault tolerant consensus-based application which runs across the network between separate instances (even safely across the Internet). This is demonstrated by our prototype demo doing just that, this was discussed in the previous section.

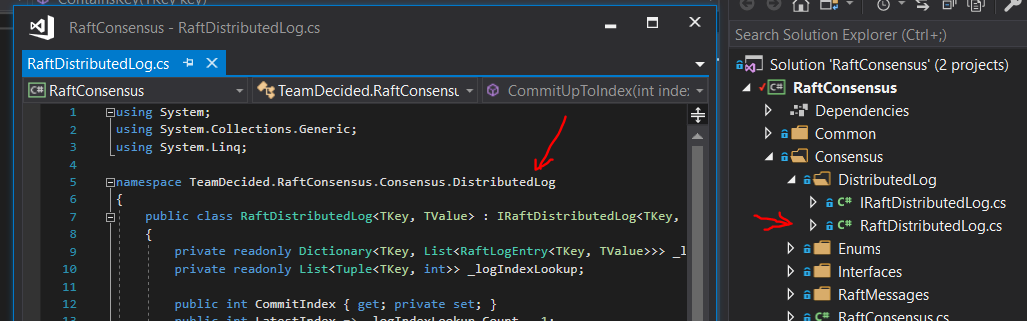
### 

### 

## Evidence of Coding Standards

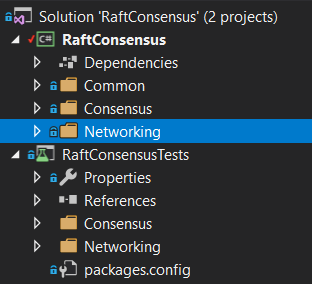
We take coding standards so seriously that we’ve ourselves defined them in our own non-functional requirements of Quality. To enforce these standards, we use the industry standard ReSharper software {website image}. This is a plugin for Visual Studios and continually runs over the code providing suggestions for coding standard compliance and naming conventions. This software has been run over our code and we’re totally consistent with its standards across the whole consensus/networking library. Even our prototype demo implements this standard as well. Here are some examples of code meeting the standards

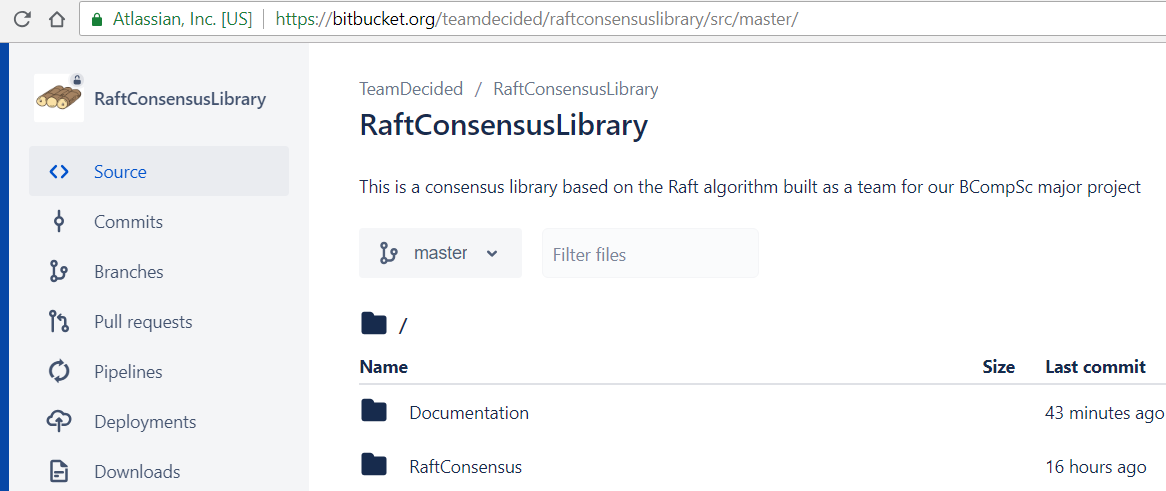


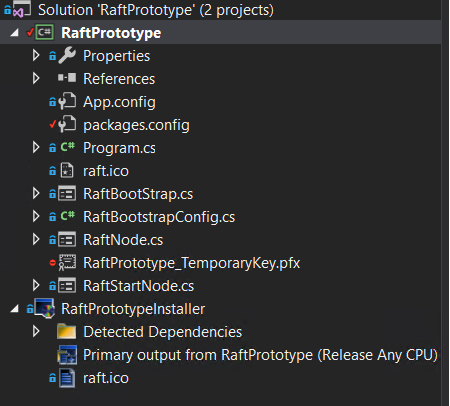


## Organization of The Development Area

As we’re developing which is functionally a headless multithreaded and asynchronous consensus algorithm communicating over the network using encryption and authentication, there are a crazy amount of moving parts. To combat these extreme levels of complexity, a huge effort has been placed in the simplicity of the development area, fighting tools is unreasonable when trying to debug complex multithreaded issues in code.

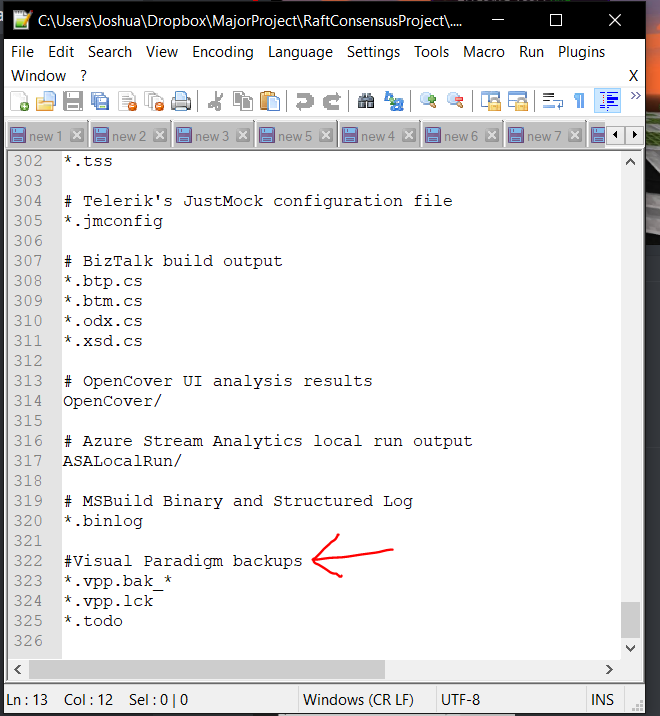
As well as where above where we’ve spoken about coding standards, we also keep the project as slim as possible having only the required projects in VS solutions and required code/classes in projects. There is nothing kept which isn’t used. Our consensus library is simply implemented in a single Visual Studio solution, with one project for the code, and one for the unit testing {image}. This library [has its own git repo on Bitbucket](https://bitbucket.org/teamdecided/raftconsensuslibrary/src/master/).



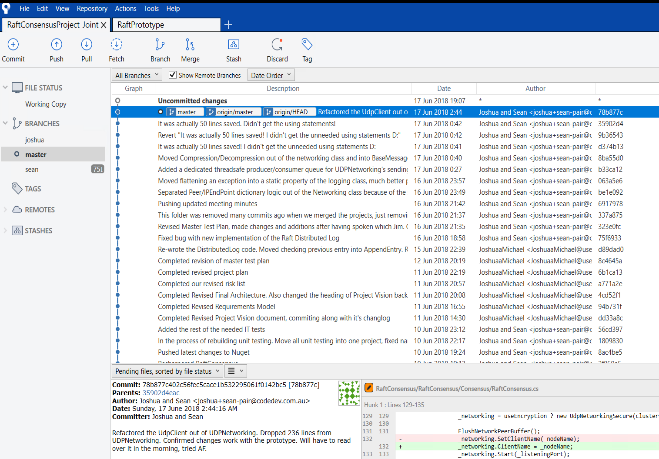


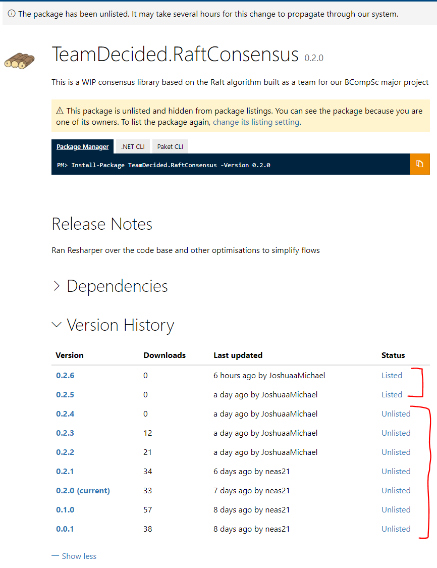
The prototype is similarly organized, only requiring two projects in it’s Visual Studio solution. One of these is for the prototype program itself, the other is used for creating/signing installers as required {image}. This project also [has its own separate git repo on Bitbucket](https://bitbucket.org/teamdecided/raftprototype/src/master/).

Each of our git repos use elaborate standard 325 line. gitignore files to ensure nothing unnecessary is committed to our repo. We’ve also added additional filtering to exclude any unnecessary files made during documentation, specifically by Visual Paradigm {image}. These files are based off of [GitHub published .gitignore](https://github.com/github/gitignore/blob/master/VisualStudio.gitignore) standard file for Visual Studios.

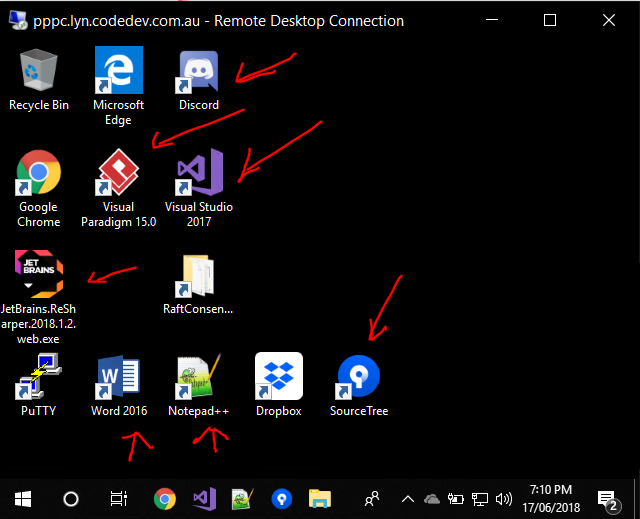


On our local computers we use SourceTree for Version Control, as it has inbuilt integration with Bitbucket. This greatly reducing complexity of using command line interface or non-first party solutions.

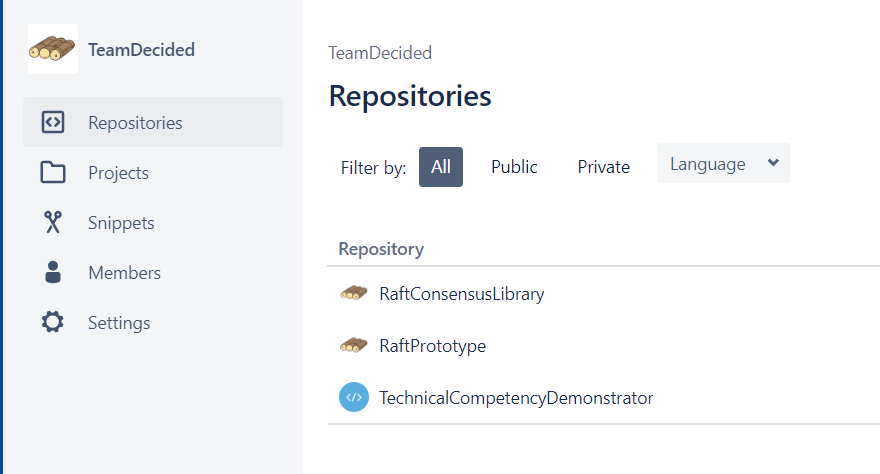


On Nuget which has the publicly accessible versions of our code, we also unlist old versions of the code which should not be used by the public {image}. This keeps it as simple and clear as possible which version may be used when implementing into a program.

For code which is developed together through pair programming, we use a dedicated virtual machine running Windows 10 inside a virtual machine hosted on spare hardware at one of our homes. One person is able to RDP into the VM, and the other comes in through remote desktop software TeamViewer, both people have active control over keyboard/mouse. This VM is a slimmed down PC with only developer tools installed like Visual Studio, ReSharper, Visual Paradigm, SourceTree, Office suite, Chrome and TeamViewer {image}. We also use a dedicated Discord server for text, voice calling, presence and push communication.

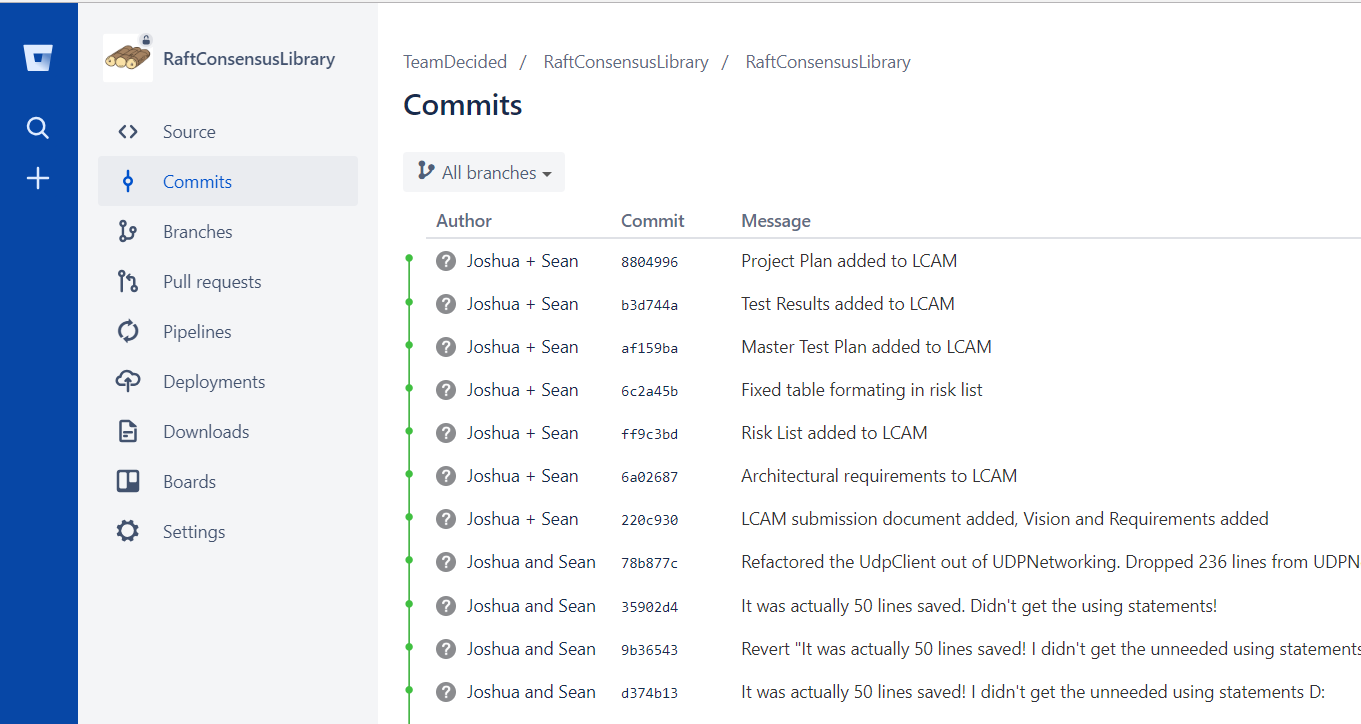


## Version Control Functional Level Commits

As described above, each project has its own separate Bitbucket repo {image}.

Also, during development, developers are doing their commits at a functional level. We follow “Commit early and commit often”. This enable the work of each developer to be made available to all members, as well as allows granularly reversion if required.

This is demonstrated in the following image of recent commit activity. {image}



## EA Implementation of All Relevant Functional and Non-functional Requirements

As supported by our Master Test Plan, and Test Results documents **{inter-document links}**. We’ve previously demonstrated that our EA implements all required functional and non-functional requirements for this stage of the project.