In today's modern world computing is becoming more fundamental to everything we do, and with that comes a focus on creating highly available and distributed services. With high availability services, many computers work together to make a service achieve the greatest uptime possible, this includes being fault-tolerant to various outage scenarios. Consensus algorithms are a foundational part of building these systems.

Raft Consensus

ITC309 – PRM
Assessment 3

Joshua Cameron;Sean Matkovich

Contents

Implementation Model

Intended Production Environment

Our project is a library which developers can use to implement consensus into their projects without having to learn/understand/implement a consensus algorithm. It aims to drastically lower the bar for developers wanting to increase the reliability of their services. Our project's intended environment is in these developer's mission critical projects as part of their goals of increasing reliability. The developers simply read our documentation, download our library through .NET's NuGet package manager, and begin integration. Part of being in these mission critical program is an important focus on reliability, which is why our most important non-functional requirements revolve around various aspects of this.

Although our library is written in a way which allows for ease of integration, the developer's projects also require a large refactoring on their end to design their program in a way which makes use of consensus/distributed consistent log, this is something that would have been required by any consensus application level library, not just our own. As it's an unreasonable effort to ask our library to be implemented in a beta tester's project, or even a simple project online, we've been maintaining our prototype project for demonstration purposes of the library working in its intended production environment. Our prototype is able to demonstrate all the features of our library, and it does so as its own self-contained project simply having downloaded our library from NuGet.

No Known Bugs

An important part of being "beta ready" is having no known bugs in the software, and the project is proud that it's software currently has no known bugs. There is an extensive 86% coverage for unit/integration testing which covers all "happy day" scenarios. This level of coverage for unit testing is walking balance between simply wasting effort on things which may be changed later and ensuring functionality. There have also been extensive hours trying to break the algorithm and reading trace level logs to ensure it's doing what it should be. This gives the project the confidence to back up the claim that it has no known bugs.

High Quality Code

A big part of reliability is usability, and code usability comes in the form of high-quality code. High quality code is not a simple process, but an iterative design process in which new ideas must be propagated through the whole system until a final product is designed. Last session for the first iteration, when starting to develop the project, an approach of “first person to touch code loses” was used, and the two weeks was instead spent in Visual Paradigm diagramming out classes and various processes. This session the project has maintained its commitment to high code quality code, and again the first iteration of the session was taken, but for refactoring this time. The refactoring effort focused firstly on avoiding multithreading deadlocks due to the many threads, secondly to separate classes doing too much into their own classes, and thirdly to reduce cyclomatic complexity across the whole codebase. This effort was successful and has led to a noticeable reduction in issues requiring debugging, debugging times, and has abolished debugging multithreading issues all together.

Evidence of Best Practice Version Control

- Library uses branches
    - Master is used for pushing to NuGet, and is treated as our production branch
    - Each developer has their own branch in which they work on code for before integration
- Commits are done often
    - Currently averaging around 10 commits per week per developer
- Tests are run before committing to master
    - Each developer ensures tests pass before committing new code to master
- Integrate frequently
    - Merges happen at least once per week, primarily more. This keeps all branches up to date and minimizes the need to resolve commit conflicts, this having only happened only twice so far in the project
- Descriptive commit messages
    - All commit messages focus on explaining why the change was made, rather than what was changed
- Single intent commits
    - All commits are required to do one thing, and this is part of already committing frequently

Feature Completion from Initial Project Aims

Results for Functional Requirements

Need	Priority	Features	Implementation	Result	Evidence
Consensus between distributed systems	**1**	Replicated log, with consensus algorithm	Feature complete implementation of the Raft Consensus Algorithm which maintains a replication consistent log amongst distributed systems	Success	• Open source • NuGet package
Fault tolerant distributed service	**2**	Consensus algorithm allows for a fault tolerant distributed system	We implemented a prototype demo which used our NuGet package, and have shown its ability to maintain a service even during node failure	Success	• Prototype
Improved reliability of existing service	**3**	System is fault tolerance, so it will improve reliability	We implemented a prototype demo which used our NuGet package, and have shown its ability to maintain a service even during node failure	Success	• Prototype
Complete proven reliability	**4**	Based on proven algorithm	Firstly, the library is based on a proven algorithm, so to ensure our implementation we've thoroughly unit and integration tested it	Success	• Testing coverage report
Minimal additional surface area for failure	**5**	Complete coverage unit testing	We've completely covered all "happy day scenarios" with our testing, this is discussed further in High Quality Code	Success	• Testing coverage report
Cross Platform	**6**	Targeting .NET standard framework	Our library is written as .NET standard 2.0 framework, which allows integration into any .NET project cross platform (pc, mobile, web, console, etc.)	Success	• Setting on project

Mitigate project abandonment	**7**	Licensing allow for profit	We've open sourced this project, so people are able to contribute or fork We're utilising the Apache2 license which allows for profit and business maintainers to continue to maintain code	Success	• Open source repo • License in repo
Minimal overhead/impact to service performance	**8**	Equivalent to leading consensus algorithm, Paxos in performance	A feature of the Raft Consensus algorithm is that it's performance is equivalent to Paxos, and messages are not wasted.	Success	• Reference thesis
Minimal resource usage	**9**	Consensus Log compaction	We've expanded on Raft Consensus's base implementation	Success	• Commit
Ability to attempt to designate a node to run the UAS	**10**	Add method to API to allow for attempting to become leader of the cluster, so as to start UAS	We agreed that this was such a fundamental change to the code base as it was such an expansion on the base implementation that we didn't have the time to produce it with our standard of existing reliability	Rejected	• Meeting minutes
Upgrade path	**11**	Versioning built in, backwards compatibility minor releases and single major	This was always an optional requirement for us, and our lowest priority. Although it's rather simple to implement (adding version number to `BaseMessage` packet class, and then filtering version in message validation method), it's low priority and simple enough someone else may implement it if it's really required	Rejected	• Meeting minutes

Results for Non-Functional Requirements

Requirements	Priority	Solution	Implementation	Result	Evidence
Reliability	**1**	1. Full coverage unit testing	We've completely covered all "happy day scenarios" with our testing, this is discussed further in High Quality Code	Success	• Testing coverage report • High Quality Code
Usability	**2**	1. Designed to be as simple as possible to integrate. 2. Released as NuGet package	We've spent two iterations so far directly achieving better usability, this is discussed more in High Quality Code above. We've released this as a NuGet package, and it's already available online	Success	• NuGet package
Documentation	**3**	1. Full coverage documentation for algorithm and API	We've produced extensive documentation which covers all use cases, gives code examples, and installation instruction. It also completes this to a level which allows	Success	• User Manual
Quality	**4**	1. Full coverage unit testing 2. Strict adherence to style guide	We've completely covered all "happy day scenarios" with our testing, this is discussed further in High Quality Code We've implemented JetBrains ReSharper Visual Studio Add In to ensure all code is written to its built in style guideline, as well as its other optimisations	Success	• Quality Code

Performance	**5**	1. Matches Paxos in performance of consensus 2. Own thread with ASYNC/non-blocking operations 3. Performance analysis	Raft Consensus Algorithm matches Paxos in performance, so no performance is lost Code has been written entirely multithreaded, with each node having at least 4 of its own dedicated threads, and there are no blocking operations. This was an optional work item, but we've been able to complete a basic performance analysis and we greatly improved performance in some cases of the algorithm so far	Success	• Raft paper snip link • Link to iteration plan
Compatibility	**6**	1. Written in .NET the second most popular language 2. Minimal dependencies 3. Written in .NET standard, cross platform 4. Designed to be as simple as possible to port languages	We've written this in .NET Standard to allow for cross platform development We've only got a popular cross platform JSON library, and Microsoft's own SQLite handler as dependencies	Success	• .NET Standard 2.0 • NuGet package
Availability	**7**	1. Can be run between servers locally or across Internet	This project uses UDP networking and due to the consensus algorithm's allowance for latency this also scales directly to being able to be run over the internet. We've shown this works, and a demo is available on YouTube.	Success	• Youtube link
Security	**8**	1. Network level authentication	All messages to/from a cluster are symmetrically encrypted with a shared secret, and communication to an encrypted cluster is not possible without the password	Success	• Secure networking

Privacy	**9**	1. Security measures to join cluster	All messages to/from a cluster are symmetrically encrypted with a shared secret, and communication to an encrypted cluster is not possible without the password	Success	• Secure networking
Scalability	**10**	1. Dynamic cluster membership, horizontal scaling	We agreed that this was such a fundamental change to the code base as it was such an expansion on the base implementation that we didn't have the time to produce it with our standard of existing reliability	Rejected	• Meeting minutes
Testability	**11**	1. Open source code, unit tests provided	This project has been made open source on Bitbucket	Success	• BB link
Extendability	**12**	1. Open source code	This project has been made open source on Bitbucket	Success	• BB link
Auditability	**13**	1. Open source code 2. Logging	This project has been made open source on Bitbucket	Success	• BB link
Troubleshooting	**14**	2. Verbose logging	This project has been made open source on Bitbucket	Success	• BB link

Demonstration

To demonstrate and explain the functionality of the library, we've put together a presentation which is available here.

User Acceptance Test

Beta Tests

For beta testing of our library it was unreasonable to get developers of mission critical software to integrate our library into their own projects as a separate solution for their consensus needs when the code base in only entering the beta stage. So, upon discussion with our lecturer we agreed that the best way to demonstrate beta level testing would be to implement it into our own demonstration program. For that, we've continued maintaining and adding our newer features into our Prototype program which utilises Microsoft's WinForms and the .NET 4.6 framework to provide the user a graphical interface to a basic text distributed key/value store. This program can be used to enable users to spin up and test nodes on their local computer, as well as used to create nodes which can talk across the internet.

The following UATs for beta match up with our library's Use Cases, and we've furthermore extended them in cases such as "survive node failure" and "Node Re-join Cluster/Rebuild" which is typically hidden from the user.

- Install prototype
- Create config
    - Without encryption, without persistent storage
    - With encryption
    - With persistent storage
    - With encryption and persistent storage
- Join Cluster
    - Load node following config creation
    - Load from persistent storage
- Append Entry
- Receive Commit Entries
- Stop Node
- Start Node
- Survive node failure
- Node rejoin cluster/rebuild
- Developer Read Log

Beta Test 1 - Install Prototype

Use Case	Install Prototype		
Test Type	Unsupervised User Acceptance Test		
Test Description			
Pre-Conditions:	User has not yet downloaded or installed Raft Prototype application		
Post-Conditions:	Raft Prototype is downloaded and installed		
Notes:	The user is expected to understand how to do the basic task of running through an installer clicking “Next” and accepting any security dialog popups.		
Results	**Pass**		
Step.	**Step Description**	**Expected Result**	**Result**
1	Download Raft Prototype from here	User download .msi file	Download completed successfully
2	Proceed through the standard steps of the installer. Accept any warning.	User installs software to computer.	Procced through windows warning Standard application installation process

Beta Test 2 - Create Config - Unencrypted/ephemeral storage

Use Case	Create Config - Unencrypted/ephemeral storage
Test Type	Unsupervised User Acceptance Test
Test Description	This is the most basic config creation test, it ensures a user is able to build a config describing an unencrypted and ephemeral cluster
Pre-Conditions:	User has installed Raft Prototype
Post-Conditions:	A Raft Consensus Config ".rcc" file is created, user is left on dialog asking if they'd like to start one of the nodes
Notes:	
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Open Raft Prototype from the shortcut made on the desktop	Raft Prototype program opens on computer	Found desktop icon
2	Select 'Create cluster config'	Takes user to Create Cluster Config page	Application started easily identified 'Create Cluster Config' button
3	Use the example configuration of: Cluster name - "My first cluster" Encryption - Leave unchecked Join Retry Attempts - 5 Persistent Storage - Leave unchecked Number of nodes - 3	User can change name Retry time auto calculates to 25s	Identified and updated with information detailed.
4	Select to 'Build' the config, and select where to save the file as "RaftClusterConfigB2.rcc"	Config builds successfully Opens dialog window File is saved to targeted location	Save dialogue box opened, successfully saved configuration file to target location.
5	Exit the application.	Close application	Exited Application by pressing the window 'X' button

Beta Test 3 - Create Config - Network encryption enabled

Use Case	Create Config - Network encryption enabled
Test Type	Unsupervised User Acceptance Test
Test Description	This is config creation test, it ensures a user is able to build a config describing a config with network encryption enabled
Pre-Conditions:	User has installed Raft Prototype
Post-Conditions:	A Raft Consensus Config ".rcc" file is created, user is left on dialog asking if they'd like to start one of the nodes
Notes:	
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Open Raft Prototype from the shortcut made on the desktop	Raft Prototype program opens on computer	Found desktop icon
2	Select 'Create cluster config'	Takes user to Create Cluster Config page	Application started easily identified 'Create Cluster Config' button
3	Use the example configuration of: Cluster name - "My first cluster" Encryption - Check, set password Join Retry Attempts - 5 Persistent Storage - Leave unchecked Number of nodes - 3	User can change name User can check Encryption checkbox User can set a valid password Retry time auto calculates to 50s	Identified and updated with information detailed.
4	Select to 'Build' the config, and select where to save the file as "RaftClusterConfigB3.rcc"	Config builds successfully Opens dialog window File is saved to targeted location	Save dialogue box opened, successfully saved configuration file to target location.
5	Exit the application.	Close application	Exited Application by pressing the window 'X' button

Beta Test 4 - Create Config - Persistent Storage enabled

Use Case	Create Config - Persistent storage enabled
Test Type	Unsupervised User Acceptance Test
Test Description	This is config creation test, it ensures a user can build a config describing a config with persistent storage enabled
Pre-Conditions:	User has installed Raft Prototype
Post-Conditions:	A Raft Consensus Config ".rcc" file is created, user is left on dialog asking if they'd like to start one of the nodes
Notes:	
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Open Raft Prototype from the shortcut made on the desktop	Raft Prototype program opens on computer	Found desktop icon
2	Select 'Create cluster config'	Takes user to Create Cluster Config page	Application started easily identified 'Create Cluster Config' button
3	Use the example configuration of: Cluster name - "My first cluster" Encryption - Unchecked Join Retry Attempts - 5 Persistent Storage - Set to checked Number of nodes - 3	User can change name User can check Persistent Storage Retry time auto calculates to 25s	Identified and updated with information detailed.
4	Select to 'Build' the config, and select where to save the file as "RaftClusterConfigB4.rcc"	Config builds successfully Opens dialog window File is saved to targeted location	Save dialogue box opened, successfully saved configuration file to target location.
5	Exit the application.	Close application	Exited Application by pressing the window 'X' button

Beta Test 5 - Create Config - Network encryption enabled/Persistent Storage Enabled

Use Case	Create Config - Network encryption and persistent storage enabled
Test Type	Unsupervised User Acceptance Test
Test Description	This ensures a user is able to build a config describing a config with network encryption and persistent storage enabled
Pre-Conditions:	User has installed Raft Prototype
Post-Conditions:	A Raft Consensus Config ".rcc" file is created, user is left on dialog asking if they'd like to start one of the nodes
Notes:	
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Open Raft Prototype from the shortcut made on the desktop	Raft Prototype program opens on computer	Found desktop icon
2	Select 'Create cluster config'	Takes user to Create Cluster Config page	Application started easily identified 'Create Cluster Config' button
3	Use the example configuration of: Cluster name - "My first cluster" Encryption - Check, set password Join Retry Attempts - 5 Persistent Storage - Set this to checked Number of nodes - 3	User can change name User can check Encryption checkbox User can set a valid password User can check persistent storage checkbox Retry time auto calculates to 25s	Identified and updated with information detailed.
4	Select to 'Build' the config, and select where to save the file as "RaftClusterConfigB5.rcc"	Config builds successfully Opens dialog window File is saved to targeted location	Save dialogue box opened, successfully saved configuration file to target location.
5	Exit the application.	Close application	Exited Application by pressing the window 'X' button

Beta Test 6 - Join Cluster - Join from loading config

Use Case	Join Cluster - Join from loading config
Test Type	Unsupervised User Acceptance Test
Test Description	This ensures that a user can load a node from a config file, and have that node join a cluster
Pre-Conditions:	User has completed one of the Beta Test 2
Post-Conditions:	User starts has a function cluster where each node has joined the cluster. User has two running nodes on their computer
Notes:	If the user takes too long starting the other nodes, they'll eventually be prompted to search again. This test is a prerequisite for the Beta Test7, two nodes must remain running.
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Using the Desktop icon, start three instances of the application.	Three instances of the application are started.	Started the application three times.
2	For each application instance open "RaftClusterConfigB2.rcc" using the Start Existing Node button.	Each application displays the Start Node window.	Opened "RaftClusterConfigB2.rcc" configuration file in each application. Three instances of Start Node window displayed.
3	Change Node selector drop-down in each Start Node window, ensure a different Node is selected in each.	Each Start Node window has different Node selected. Port numbers are also different.	Window one displays Node1 Window two displays Node2 Window three displays Node3
4	Start each node by pressing the 'Start Node' button.	Each node is started displaying a Node Detail window.	Window one displays Node Detail window Window two displays Node Detail window Window three displays Node Detail window
5	Exit the application running Node1.	Application window running Node2 and Node3 continue to run.	Exited Application by pressing the window 'X' button Two running windows remain open

Beta Test 7 - Join Cluster - Join from creating config

Use Case	Join Cluster - Join from creating config
Test Type	Unsupervised User Acceptance Test
Test Description	This tests that a user can follow on from creating a config to running their first node, to starting more nodes to create a cluster
Pre-Conditions:	User has completed a Beta Test 6
Post-Conditions:	User has been able join cluster
Notes:	If the user takes too long starting the other nodes, they'll eventually be prompted to search again
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Beta Test 6 was completed successfully	Applications running Node2 and Node3 are active.	Application for Node2 and Node3 are running
2	Follow Beta Test 2 steps 1 through 3	User has changed cluster name Retry time auto calculates to 25s	Completed steps 1 - 3
3	Select to 'Build' the config, and select where to save the file as "RaftClusterConfigB7.rcc"	Config builds successfully Opens dialog window File is saved to targeted location	Save dialogue box opened, successfully saved configuration file to target location.
4	Select Node1 from Node selector drop-down in Start Node window. Start node by pressing the 'Start Node' button.	Node1 is selected in Start Node window. Node1 Node Detail window is loaded	Node1 Selected in Start Node window Node Detail window for Node1 opens.
5	On application running Node1 select the Debug Log tab.	Server status is "Follower" Debug log contains "We've found the cluster!" entry	Debug Log tab show server status Log contains entry.
6	Collective Nodes form and create cluster	One of the nodes will show their UAS is started	Node shows Leader status

Beta Test 8 - Append Entry

Use Case	Append Entry
Test Type	Unsupervised User Acceptance Test
Test Description	This ensures that a user can attempt to commit a new entry into the distributed log
Pre-Conditions:	User has created a cluster of any type
Post-Conditions:	User submits an entry to be committed to the log
Notes:	Appending Entries are really requests to append, and are not guaranteed. Nonetheless in a cluster running so on the same machine the log will be updated almost instantly as the underlying algorithm does its work. For the sake of this use case we aren't including seeing it committed.
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Complete Beta Test 6 or 7	Three application running as cluster	Three applications running cluster
2	Find the Node with their UAS Showing as Running	User can identify which node is running the UAS	Node identified as running the UAS
3	At the bottom of the window type into the Key textbox "Hello", and the Value textbox "World"	User is able to type into the textboxes	Added details into the two textboxes
4	Select Append	The values disappear, a request shoots off in the background to append this entry	Values disappeared

Beta Test 9 - Receive Commit Entries

Use Case	Receive Commit Entries
Test Type	Unsupervised User Acceptance Test
Test Description	This ensures that a user can observe their entry committed across the cluster
Pre-Conditions:	User has created a cluster of any type and has just pressed Append to append a new entry
Post-Conditions:	User can observe the entry becoming committed across the cluster
Notes:	
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Complete Beta Test 8	Three application running as cluster	
2	Look at all the open nodes and observe in their log that “Hello”/”World” has appeared	“Hello”/”World” appears in the nodes logs	Append entry message appears in all applications

Beta Test 10 - Stop Node

Use Case	Stop node		
Test Type	Unsupervised User Acceptance Test		
Test Description	This ensures that a user is able to hard stop a node from participating in the cluster, emulating an unexpected hardware failure of a node to the rest of the cluster		
Pre-Conditions:	User has created a cluster of any type		
Post-Conditions:	User is able to stop a node from participating in the cluster		
Notes:	This use case only covers the behaviour of the node stopped, observing the surviving cluster will be handled in another use case		
Results	**Pass**		
Step.	**Step Description**	**Expected Result**	**Result**
1	Complete Beta Test 9	Three application running as cluster	Application running with committed entry
2	Select “Stop” on one of the nodes in the cluster	Node changes to stopped state, perhaps having stopped their UAS to get there depending on which nodes was stopped by the user	Node stopped, log data has disappeared and the start button has been enabled

Beta Test 11 - Start Node

Use Case	Start node
Test Type	Unsupervised User Acceptance Test
Test Description	This ensures that a user is able to start a node and have it join an existing cluster, emulating recovery from a crashed state
Pre-Conditions:	User has created a cluster of any type, use has stopped the node in question
Post-Conditions:	User is able to start a node and have it join an existing cluster
Notes:	
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Complete Beta Test 10	Three application running as cluster	Applications running with stopped node
2	Select “Start” on a node in the cluster what was previously Stopped	Node starts up, joins cluster and sets it status to “Running”	Started node, log has been rebuilt and the stop button is now enabled

Beta Test 12 - Survive Node Failure

Use Case	Survive Node Failure
Test Type	Unsupervised User Acceptance Test
Test Description	This use case shows that a cluster can survive a node failure
Pre-Conditions:	User has created a cluster of any type
Post-Conditions:	Cluster survives node failure
Notes:	This could also be done on a non-UAS running node, however the nodes do not react to a simple follower failing. Nothing to observe about cluster change, so it's covered by Stop Node use case.
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Complete Beta Test 9	Three application running as cluster	Application running with committed entries
2	User uses the Stop Node use case to stop the node running the UAS	The node stops running it's UAS The other remaining nodes are seen to 'elect' a new leader among them to run the UAS	Stopped active UAS, log data disappeared. Another application changed to Active

Beta Test 13 - Node re-join/rebuild distributed log

Use Case	Node re-join/rebuild distributed log
Test Type	Unsupervised User Acceptance Test
Test Description	This use case shows that a node can re-enter a cluster, and bring its log up to date
Pre-Conditions:	User has created a cluster of any type, user has committed some entries to the log, user stops any node
Post-Conditions:	Node re-enters the cluster and has its log brought up to date
Notes:	
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Complete Beta Test 12	Three application running as cluster	Application running with committed entries
2	User uses the Start Node use case to bring back a previously stopped node into the cluster	Node finds cluster, the node running the UAS immediately starts populating the log of the previously missing node and brings it up to date	Node restarts and committed log is updated

Beta Test 14 - Read developer logs

Use Case	Read developer logs
Test Type	Unsupervised User Acceptance Test
Test Description	This use case to show off the levels of logging the raft algorithm is doing in the background
Pre-Conditions:	User has created a cluster of any type, user has committed some entries to the log
Post-Conditions:	User is able to observe developer logs
Notes:	This information can be useful for debugging any issues that may occur. The debug level is set when using the code, and defaults to debugging at info level
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Complete Beta Test 12	Three application running as cluster	Application running with committed entries
2	User picks any nodes and selects to move over to the Debug Log tab	User can view and scroll through the back-end developer log of the consensus algorithm	Debug log tab selected, debug log contains information
3	Use selects Debug or Trace from the dropdown and sees the flow of messages of flying by that the algorithm is writing to the log for debugging purposes	User can view and scroll through the back-end developer log of the consensus algorithm, they can now see heartbeat and message flow/processing	Trace level debug log displays extremely detailed amount of message traffic Debug level debug log displays large amount of message traffic

Beta Test 15 - Node start from persistent storage

Use Case	Load from persistent storage
Test Type	Unsupervised User Acceptance Test
Test Description	This use case to show off with the prototype that it successfully implements being able to load a previous log from persistent storage
Pre-Conditions:	User has created a cluster with persistent storage enabled, all 3 nodes are connected, and 5 entries are committed
Post-Conditions:	User is able to stop and start a node, and does not need to get rebuilt on start up since it already has the values
Notes:	Losing a node is simulated with the start/stop buttons, so this is a valid test of it the data is still available after a restart of the node
Results	**Pass**

Step.	Step Description	Expected Result	Result
1	Using the Desktop icon, start three instances of the application.	Three instances of the application are started.	Started the application three times.
2	For each application instance open “RaftClusterConfigB4.rcc” using the Start Existing Node button.	Each application displays the Start Node window.	Opened “RaftClusterConfigB4.rcc” configuration file in each application. Three instances of Start Node window displayed.
3	Change Node selector drop-down in each Start Node window, ensure a different Node is selected in each.	Each Start Node window has different Node selected. Port numbers are also different.	Window one displays Node1 Window two displays Node2 Window three displays Node3
4	Start each node by pressing the ‘Start Node’ button.	Each node is started displaying a Node Detail window.	Window one displays Node Detail window Window two displays Node Detail window Window three displays Node Detail window
5	Find the Node with their UAS Showing as Running	User can identify which node is running the UAS	Node identified as running the UAS

6	Use the Append Entry use case to add a series of messages to the consensus log	User can type into the textboxes And Appends several entries to the log.	Added details into the two textboxes Values disappear, the entry is committed and is displayed in all logs
7	User uses the Stop Node use case to stop the all node running	All nodes stop All Consensus logs are emptied	All nodes stopped
8	Restart all node in random order	All nodes restart Each nodes consensus log is populated with details from persistent log.	Nodes re-join cluster All consensus logs are returned to former state

Programmer Documentation

Business aims

Create a fully featured open source code library to allow the most amount of developer's access to adding consensus features as easily as possible to their projects

Consensus

From the mission statement, fundamentally this project is first and foremost about producing a consensus library to lower the barriers for developers to add high availability fundamentals into their projects. This is the primary focus and functionality of this library which needs to be maintained. It's always prudent to remember the people implementing this library are trusting in the development of it, as they'll be implementing it into their most mission/business critical applications.

Based on proven algorithm

It goes without saying, consensus is a complicated beast, and as such requires a large investment of developer time to truly understand the ins and outs. Although there are alternative consensus algorithms available such as Paxos, it's an open secret that Paxos is an extremely complex algorithm. As complexity is the enemy of reliability this library currently implements the Raft consensus algorithm due to its usability/simplicity focused design and coupled with its equivalent performance to Paxos it's the perfect foundational algorithm for this project. More information about the algorithm can be found here, and the reference paper for the implementation is more specifically found here.

Cross Platform

It's important that a library such as this be made available to the widest audience it can, and as such Microsoft's .NET Standard has been used to implement this algorithm because it's allows a "write once, run anywhere" style development, as .NET Standard works on all major operating systems and mobile devices.

Open source

It's important that a component aiming to be made a critical part of anyone's services be open and auditable as well as allow for growth and further development. As such, this library is available as Apache2, and anywhere it's released there will be references made showing potential developers that the source code is available and the project open source.

Usability

Our primary audience is developers who want to implement the extremely complex feature of consensus into their application *easily*. Therefore, there is a large focus to make the 'front end' user facing side of the code as simple as possible for developers to implement, as such already a considerable time as been spent ensuring the fewest lines of code reasonable on our part are required to integrate.

Security

As this library may be used to transfer mission critical information, specially across networks and potentially the internet, it's important that communication is secured against interception. Currently the library implements the industry standard AES cipher using a pre-shared key. There is absolutely room for this to be improved, however the currently implementation has a reasonable level of security.

Troubleshooting

This library's purpose of being integrated into mission critical application makes it imperative that developers can initially troubleshoot and diagnose any issues in the library they have themselves. There are multiple logging levels implemented into the software to enable this; it even goes as far to enable trace level logging, so command execution can be monitored to narrow down causes of any issues as easily and accurately as possible. Verbose logs are the friend of complex systems.

System Architecture

The following list of 10 architectural components making up this system are listed and discussed below, in addition to that is an architectural diagram which shows the relationship between these components and their placement in the overall system.

Architectural Diagram

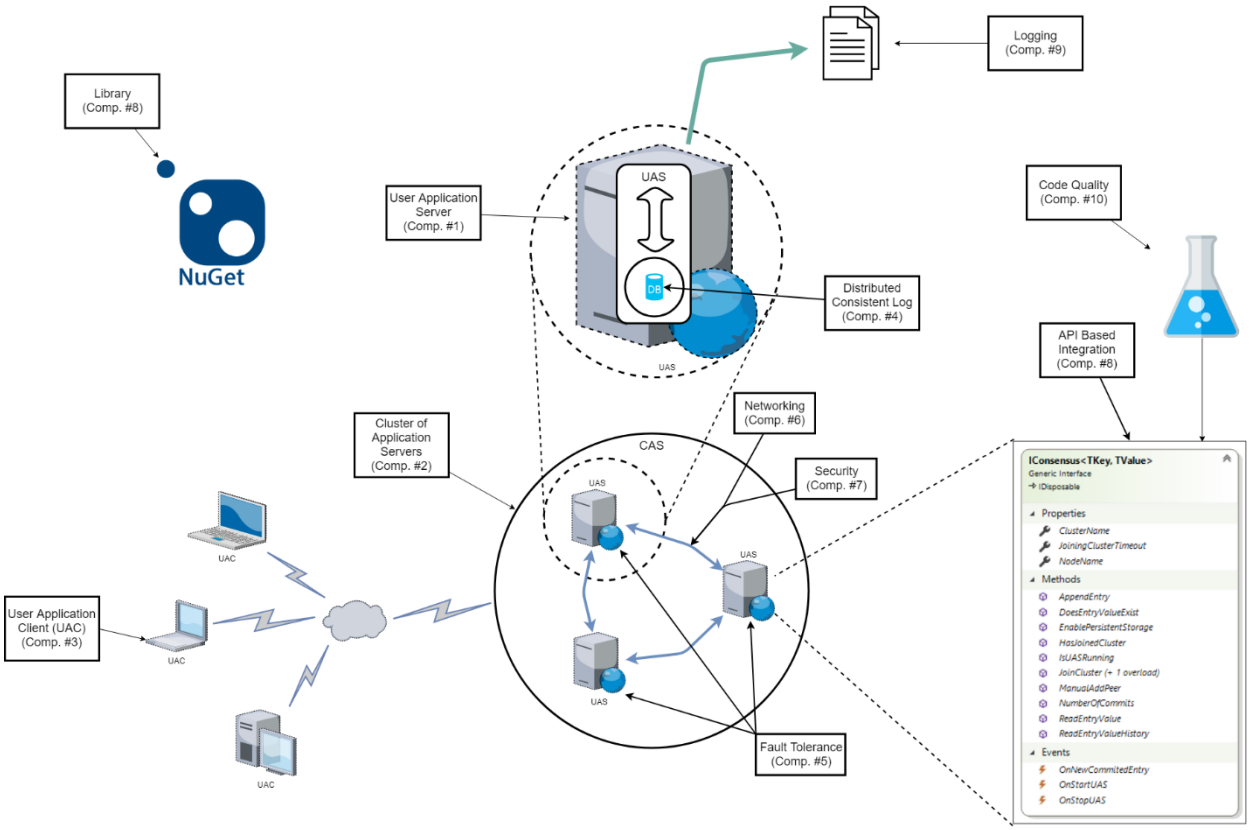

Figure 1 Overview of Architectural Components

Architectural Components

Architectural Component 1 - User Application Server (UAS)

This is the component of the architecture which is integrated into the developer's existing program. The developer is then able to run multiple instances of their software for failover and can use the library to keep them in sync. There are architectural changes required by the developer to achieve this, however those are the same considerations for any application level consensus library. This includes things like considerations about the architecture of the failover, how users will maintain access to a server after failure, how many servers in the cluster respond to requests, etc. There are multiple ways the UAS can be implemented within a developer's software. One way is by keeping each separate full server instance of their software in sync, where the developers can use distributed log feature of the UAS to allow each server to respond user requests with committed data. Another way is in a failover scenario where although all nodes in the cluster implement the full UAS + server architecture, there is only one active UAS at a time, where the other instances simply receive log entries and start their full server in the scenario of existing server failure. The implementation of how this component of the architecture is used is up to the individual's projects utilising the library.

Architectural Component 2 - Cluster of Application Servers (CAS)

This is the logical structure of the group of UASs running the cluster and is used to keep the developer's initial/target *service* highly available. UACs (discussed next) maintain contact with the CAS using a feature such as simple IP failover. This failover method is implemented into the UACs by the developers and is outside the scope of this project.

Architectural Component 3 - User Application Client (UAC)

The UAC is the User Application Client, this is client side of the developer's existing architecture which receives services from the CAS. Using an example of the CAS being a multiplayer video game server, the UAC would be players. The player's clients would be configured which each IP address of UASs in the CAS and could use simple IP failover if their target one stops responding.

Architectural Component 4 - Consensus/Distributed Consistent Log

This is the component used by the User Application Server to commit their running service's data into a distributed log amongst the consensus nodes. This is the foundational feature which allows other UASs to stay synchronized and survive node failure.

Architectural Component 5 - Fault Tolerance

This is the features which allows an increase in availability of a given service, it does this though the consensus algorithm synchronising of the distributed consistent log which can be used by another working UAS to continue offering services.

Architectural Component 6 - Network Communication

This is the functionality which allows the distributed consensus nodes to communicate with each other. It is based on the "fire-and-forget" or "connectionless" UDP protocol to reduce consensus latencies through reduction of required roundtrips, while leaving the overhead of handling packet loss to the consensus algorithm.

Architectural Component 7 - Security

This feature provides network security through encryption of those networking messages. It currently uses the industry standard AES cipher for this, with the key requiring to be both pre-shared for encryption/decryption.

Architectural Component 8 - API Based Integration

The User Application Server (UAS) communicates with CAS through the use of a .NET Standard class library. This single interface focused on usability is how the UAS communicates to the network of nodes running the consensus algorithm together. This library is also available through .NET's package manager NuGet; this is to ensure a reliable smooth integration experience for developers.

Architectural Component 9 - Logging

This is the component responsible for producing intelligible actional logging information for developers and anyone troubleshooting the works of the software. As consensus is inherently a complex system, being able to clearly understand a potential situation which lead to an issue with a company's mission critical service is very important.

Architectural Component 10 - High Quality Code

There has been a dedicated and focused effort on ensuring the highest possible quality of code as part of this project. As this code is to be ideally used in ensuring high availability, it's focus on quality must be paramount.

Detailed Design / Operations

To enable a following developer to gain a rapid and accurate understanding of the software for the reasons of maintaining it, below will be discussion about each of the major components of the system. They will be discussed from simplest to most complex, building fundamentals to further understanding as the document continues. However, please note, that an understanding of the Raft Consensus algorithm itself can no better be expressed here than from the reference paper itself, so that will not be reiterated here, but it will be assumed knowledge beyond this point. The components which will be discussed are:

- PCQueues
- WaitLoop
- Raft Message Processing
- Consensus
- IUDPNetworking
- Logging
- Unit/integration testing
- NuGet

PCQueues

The producer consumer queue pattern is a thread synchronization computer science fundamental and is used in multiple places within the library. This pattern is a solution to ensuring producers of data can effectively hand those off to consumers without any race conditions occurring. In addition to the standard implementation, we've also added a `ManualResetEvent` which triggers when there are entries in the queue; this wait handle can be waited on by the consumers, so they may only need to wake upon having entries to process. Examples of where this is used is queuing messages between producing threads for delivery over the network, receiving of messages from the network and providing to a de-serialisation and processing thread, the providing of received consensuses messages to the consensus algorithm to process.

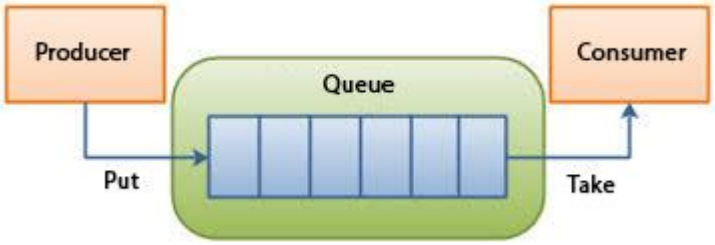

Figure 2 Producer Consumer Queue Illustration

WaitLoop

The WaitLoop is the harness behind the Raft Consensus class which enables the sleeping/waking of the background thread as work is required to do. This is a very CPU efficient method of managing the background thread and is also designed to be as simply as possible to reduce surface area for coding issues. To reduce complexity in the Raft class, the WaitLoop has been separated into its own class. The Raft background thread instantiates the WaitLoop, registers the Raft logic ManualResetEvent flags with their corresponding function (Func), and then simply executes the blocking `Run` command. This can be seen below in the sequence diagram. There is also the ability to add a Func which supports waiting a specified time, so this is what's used by leaders to timeout to send heartbeats and following/candidates to change to/restart the candidate state.

The `Run` command itself is essentially a loop which waits for flags or the timeout, then executes the code associated with the flag or timeout. Figure 3 below shows a code exert, line 56 decides which flag has triggered, or if the timeout has occurred, catches any exceptions which occur, and then line 66 is a switch statement which figures out with Func needs to be executed.

```
46          public void Run()
47          {
48              WaitHandle[] waitHandles = _waitHandles.ToArray();
49              Stopwatch stopwatch = new Stopwatch();
50              while (true)
51              {
52                  int index;
53                  try
54                  {
55                      stopwatch.Restart();
56                      index = _timeoutFunc != null
57                          ? WaitHandle.WaitAny(waitHandles, TimeoutMs)
58                          : WaitHandle.WaitAny(waitHandles);
59                  }
60                  catch (Exception e)
61                  {
62                      if (_exceptionFunc?.Invoke(e) == true) return;
63                      continue;
64                  }
65
66                  switch (index)
67                  {
68                      case WaitHandle.WaitTimeout when _timeoutFunc?.Invoke() == true:
69                          return;
70                      case WaitHandle.WaitTimeout:
71                          continue;
72                      default:
73                          {
74                              if (_funcs[index](_waitHandles[index], (int)stopwatch.ElapsedMilliseconds)) retur
75                              continue;
76                          }
77                  }
78              }
```

Figure 3 Code excerpt showing the Run() function.

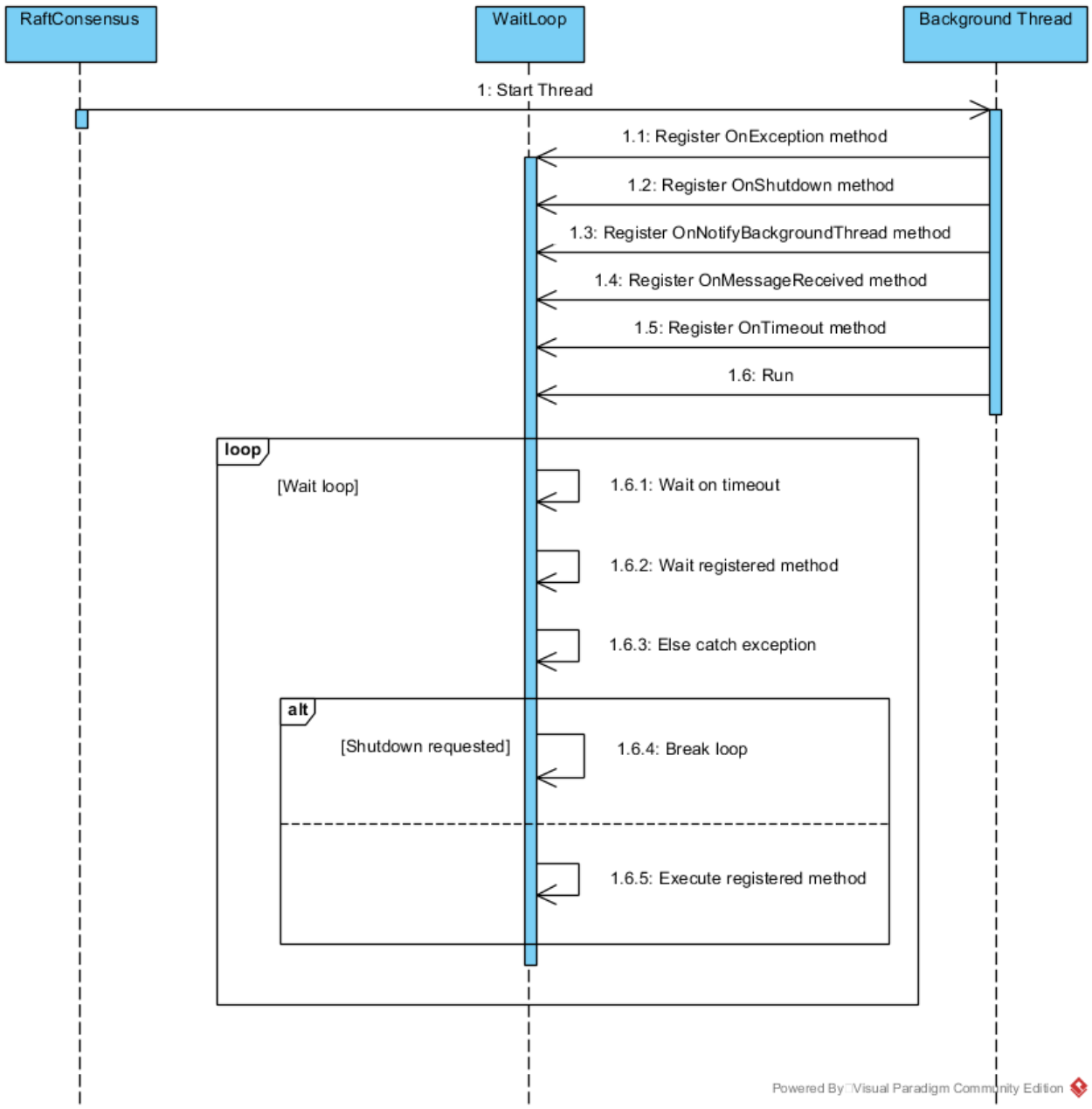

Figure 4 WaitLoop Sequence Diagram

Raft Message Processing

Each Raft message goes through a series of steps before being eventually being processed by one of the four Raft algorithm message handling methods from the specification (i.e. `HandleRaftAppendEntry`, `HandleRaftAppendEntryResponse`, `HandleRaftRequestVote`, `HandleRaftRequestVoteResponse`). As the sequence diagram below shows, the message starts off being received by the network stack (This will be further discussed in depth later), the network stack executes the method the Raft class has registered with it to handle new message, this method in Raft simply enters the message received from networking into a member variable PCQueue. Entering this message into the PCQueue naturally triggers the flag which has been registered with the WaitLoop (discussed above) to process the message, and execution on the message by the background thread begins. The Func registered with the WaitLoop then checks basic things like cluster name, and if the message uses a greater term number, then it uses a 'filter' class which uses a basic key/value map to determine whether this message can be actioned in the current Raft state. If so, the code then follows on to use another key/value store class to lookup which method has been registered to handle this type of method. Although seemingly over complicated for this level, this structure is separately testable, and allows for the simple linear scaling of the message types processed by Raft for further features in the future.

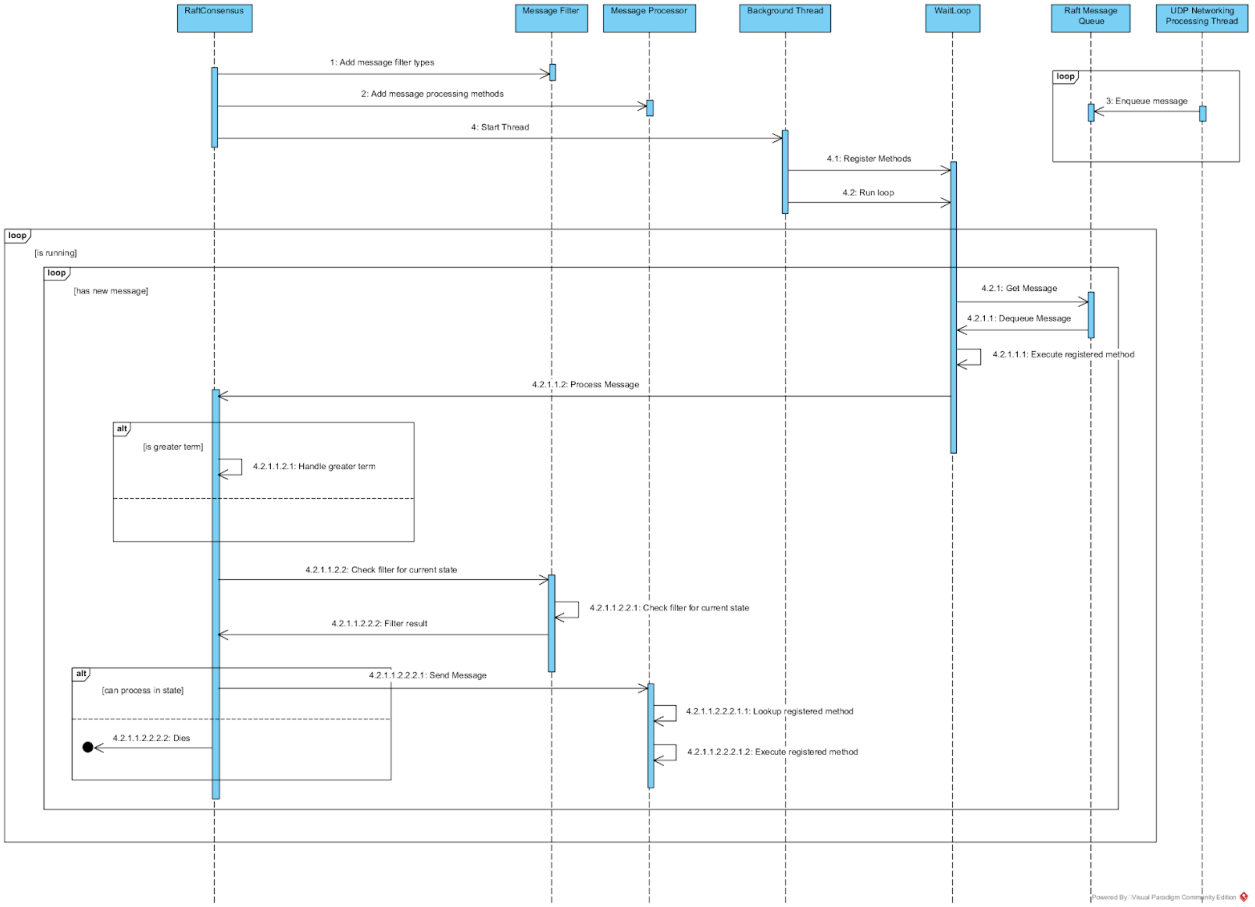

Figure 5 Message Processing Sequence Diagram

Consensus

The consensus classes are the heart and soul of this library. We go beyond the basic Raft specification of simply maintaining a list of elements for the distributed log, we maintain a generic key/value store taking on user defined types of both the key and value. This is part of the usability features for developers, ensuring it can be more widely implemented and tailored for their specific situation. The task of each of the elements defined in the class diagram below are:

- `IConsensus<TKey, TValue>` - The interface in front of any consensus algorithm implemented by the library
- `RaftConsensus<TKey, TValue>` - The Raft Consensus algorithm implementation which implements the IConsensus interface.
- `MessageProcessor<T>` - A basic key/value class which is used for mapping the message types to methods which will 'handle' them
- `WaitLoop` - The class which manages the scheduling of the background thread of the `RaftConsensus` class
- `StateMessageWhitelistFilter` - A basic key/value class which is used by the Raft message processing to determine if a given message may be processed in the current state
- `SQLiteWrapper` - A wrapper for basic SQLite operations, this is used by the Raft Log which implements persistent storage
- `RaftBaseMessage` - The base class of the four reference Raft message types
- `IRaftDistributedLog<TKey, TValue>` - An interface which is used by the distributed logs of the Raft algorithm to store the consistent data in dictionaries/maps. There are two types of these implementations (not shown), one is the ephemeral in-memory database, and the other is a persistent SQLite database implementation.
- `RaftLogEntry` - The atomic unit of a `IRaftDistributedLog`, this maintains information such as term and index numbers as required by the Raft algorithm specification
- `NodeInfo` - Each Node in the CAS keeps basic information about each other node, such as message timeouts and latest response times
- Various enums which describe response codes or states

The `RaftConsensus` class consists of methods to join a cluster/append messages/receive notices of new messages, a background thread which handles messaging and time dependant actions, and various private methods to facilitate these actions. When a node becomes or loses leadership of the cluster the corresponding `StartUAS`/`StopUAS` event is called, this notifies the developer which of their nodes is currently authoritative and enables them to implement a highly available service which has only a single running UAS.

Figure 6 RaftConsensus Class Diagram

IUDPNetworking

As the nodes of the CAS need to communicate, we needed a solution for their networking connection. After not having much luck finding a suitable open source UDP networking library which did not come with a large amount of bloat, we implemented our own minimalist library. IUDPNetworking is a interface to this networking functionality, which enables the easy migration to another networking library in the future if required. But for now, we're working with our `UDPNetworking` implementation.

The `UDPNetworking` implementation is made up of 3 background threads (the listening thread, the sending thread, and the processing thread) and events which fire under circumstances (`NewConnectedPeer`, `MessageReceived` and various errors). The listening thread is tasked with simply listening to the bound network adapters and quickly passing the Tuple of received IP address and raw message bytes to the processing thread through a PCQueue before getting back to listening again. The processing thread is notified by this PCQueue's flag, it decompresses and deserializes the received message, and then hands it over to the handlers registered with its `OnMessageReceived` event. The processing thread also first executes a virtual function which is passed the deserialized message and is utilised by extensions of the class, in our case it's used by the `UDPNetworkingBasicSecurity` class to handle decryption of an inner message before returning the plain text result for the event.

`UDPNetworking` has a method called `SendMessage` which is used by Raft to send messages to other nodes. This message serialises and compresses the given message and then passes it off to PCQueue for the `SendingThread`. UDPNetworkingBasicSecurity extends this processing by encrypting the message. The `SendingThread` wakes upon the PCQueue flag and sends it off to a helper class which looks up the destination and sends the message out the interface.

`UDPNetworking` also has helper classes, those will be discussed as below:

- `CrypoHelper` - A class which implements reference versions of Microsoft's AES implementation. These Microsoft implementations can also take advantage of CPU's dedicated AES-NI instruction set for hardware offloading of the AES encryption for further performance gains.
- `NodeIPDictionary` - A basic key/value store which is maintained as a lookup reference for the correlation between network node names, and their IP addresses. This is used to make network communication simpler, and abstractions IP addresses away from the user.
- `RaftUDPClient` - This is where the actual serialisation/de-serialisation, compression/decompression, message validation, name to IP lookups, and sending/receiving of messages over the network interfaces occurs.

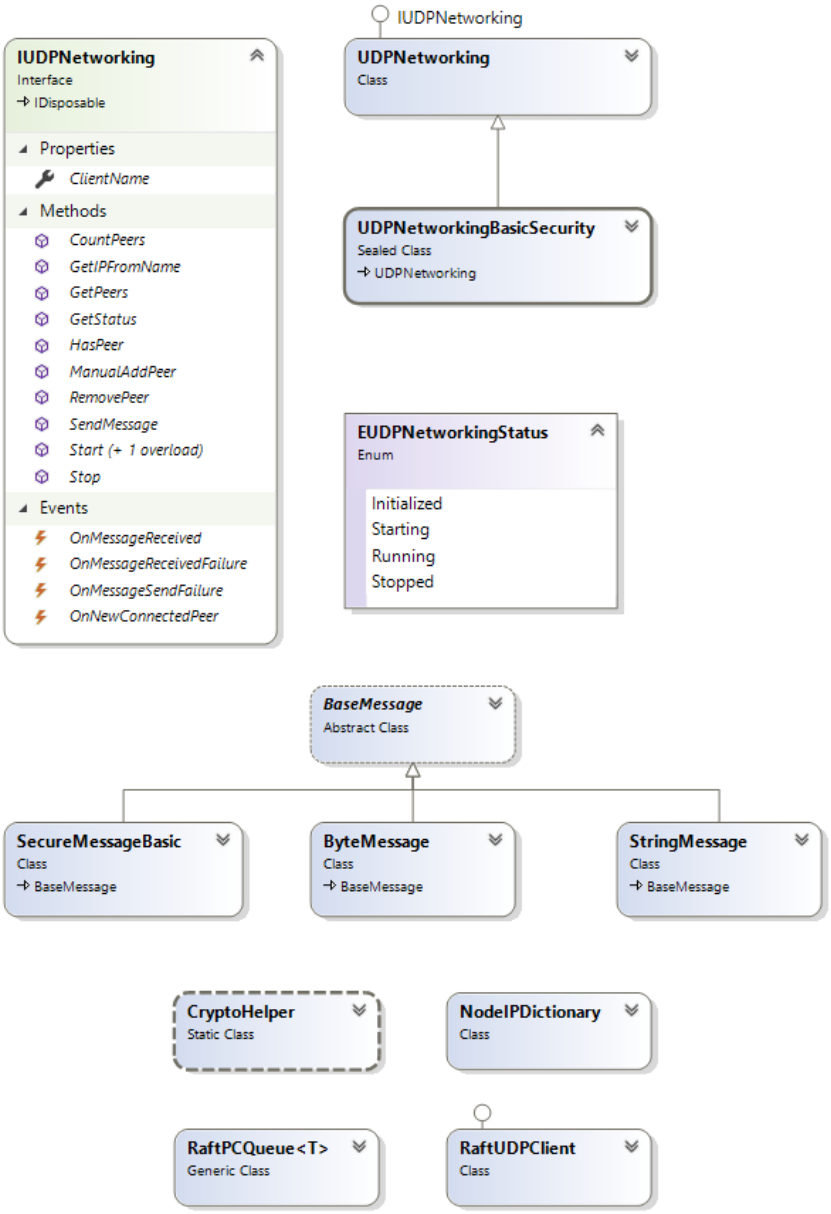

Figure 7 UDPNetworking Class Diagram

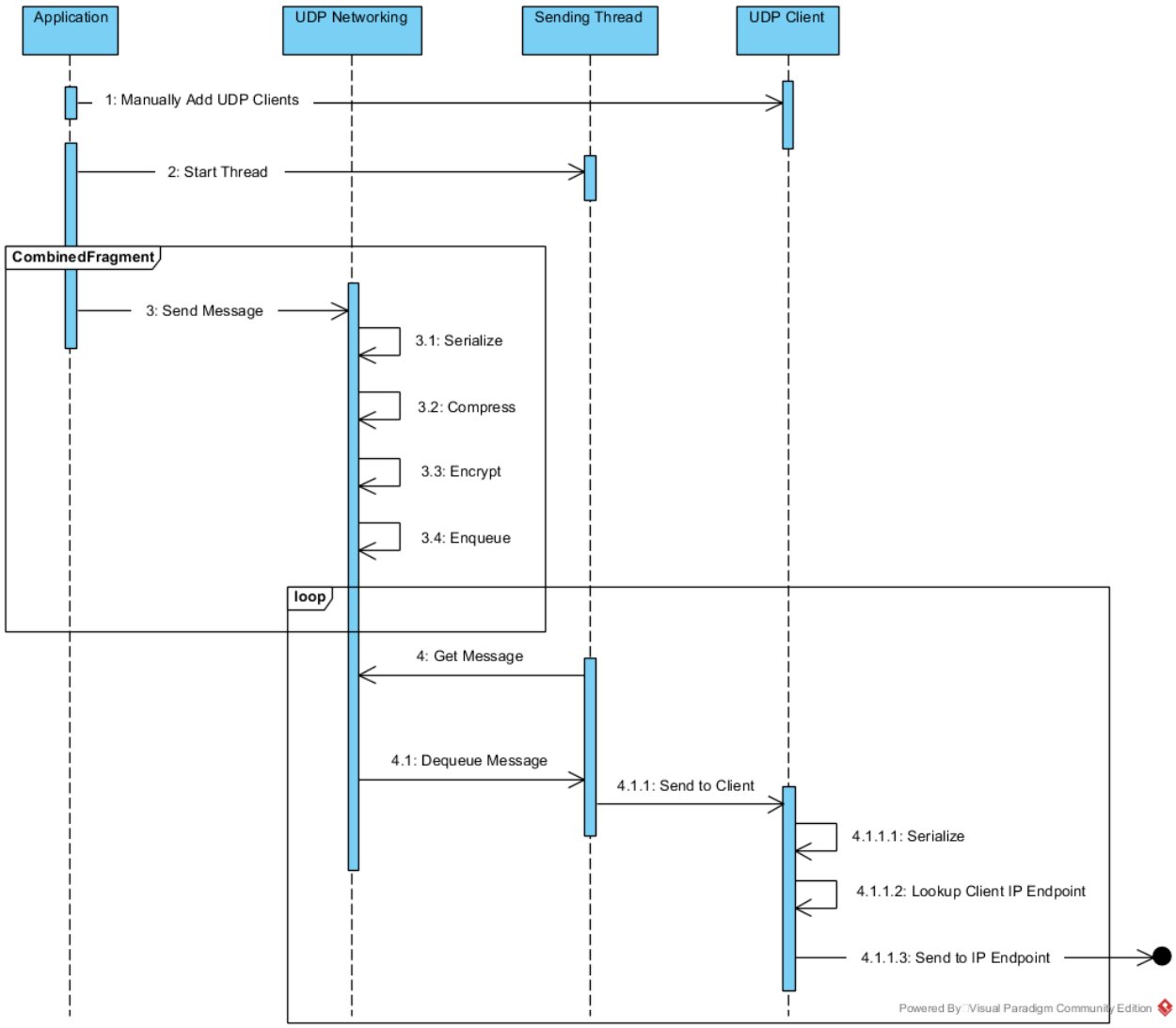

Figure 8 UDPNetworking Sequence Diagram

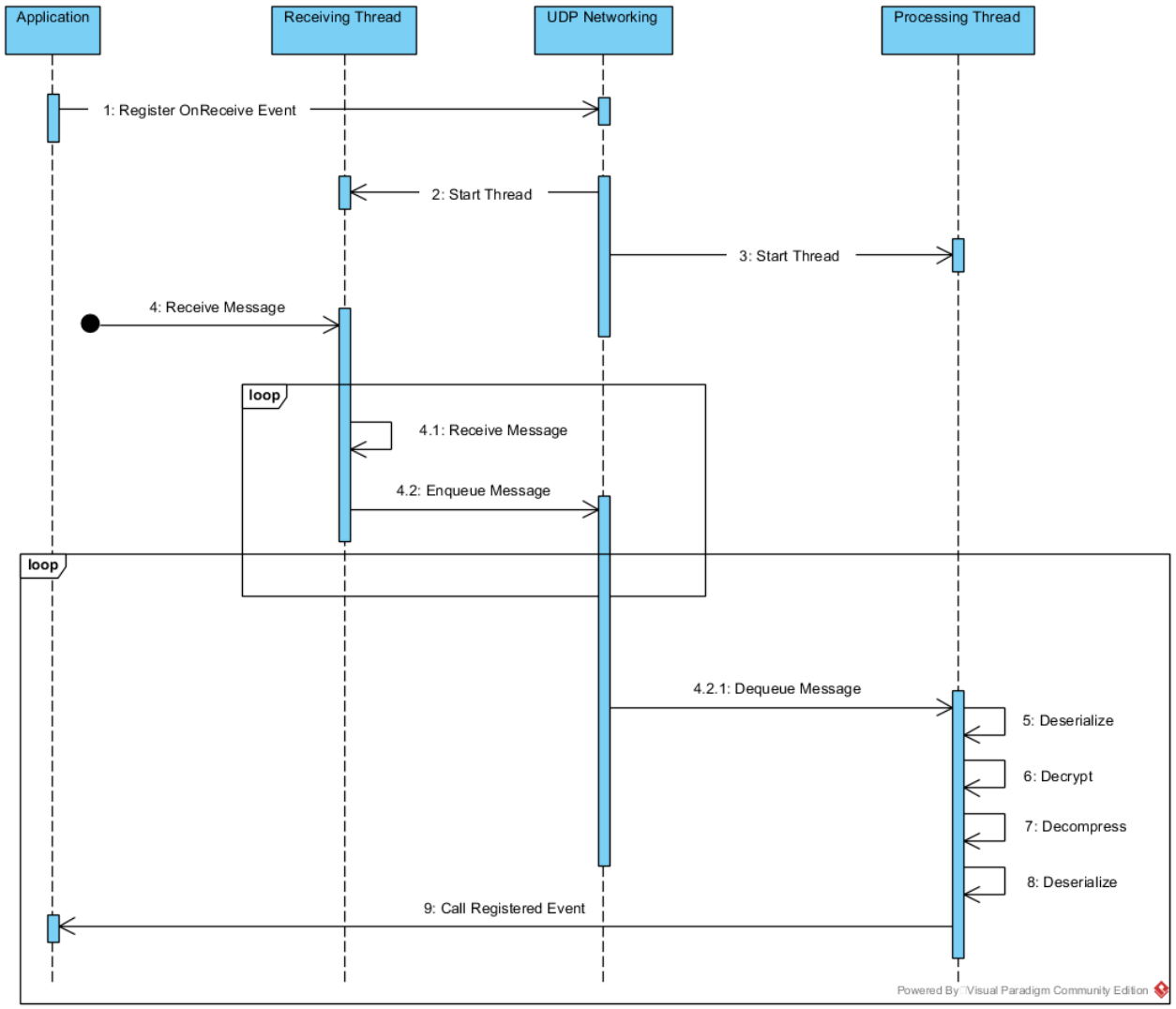

Figure 9 Messaging Sequence Diagram

Logging

Debugging multithreading issues are some of the most difficult programming challenges a developer can face, this library also has added complexity of consensus and millisecond accurate timings as well. As such, mature logging capabilities are absolutely essential to facilitate development and troubleshooting bugs. The library supports the six standard levels of logging (critical, warning, info, debug, debug and trace) and adheres to the standard usages of each level. The debugging level may be specified, and any entry which is equal to/higher will be discarded.

The logging class in implemented using a singleton pattern, must be configured with various settings before use and supports 3 forms of logging. These forms will be discussed here and followed by the various settings to be configured before use.

- Logging to event - This allows developers to tie into the logging system and get the stream of logging entries. The logging system simply calls methods subscribed to its `OnNewEntry` event.
- Logging to disk - Quiet a standard setup, but unfortunately at noisy debugging levels the IO wait of writing out to disk can disrupt the timing accuracy of the algorithm while it has to wait. To offset this issue, there is also buffering capabilities which stores up message before writing out, however naturally the buffer is lost in the singleton if an unhandled exception is thrown, and with the buffer having critical information required for debugging this can be unacceptable

- Logging to named pipe - Named pipes are an OS feature which allows separate processes to talk to one another. Typically for use in development and when trace level logging is required for difficult issues, the library will send each entry across to a totally separate program which listens to the other end of the named pipe. This is an internal through RAM transfer between processes, and as such is unbelievably fast. It enables the viewing of timings 1000 times more accurate than millisecond and maintaining logs upon crashes. Our implementation buffers messages before periodically flushing them out to disk. This reference software is also available with the RaftConsensus Library project files.

The below code example also shows basic commands for setting up the logging class to write to NamedPipe, File, and Event. You do not technically need to setup any of them, and nothing will be logged, however it's obviously recommended to setup at least one form of logging.

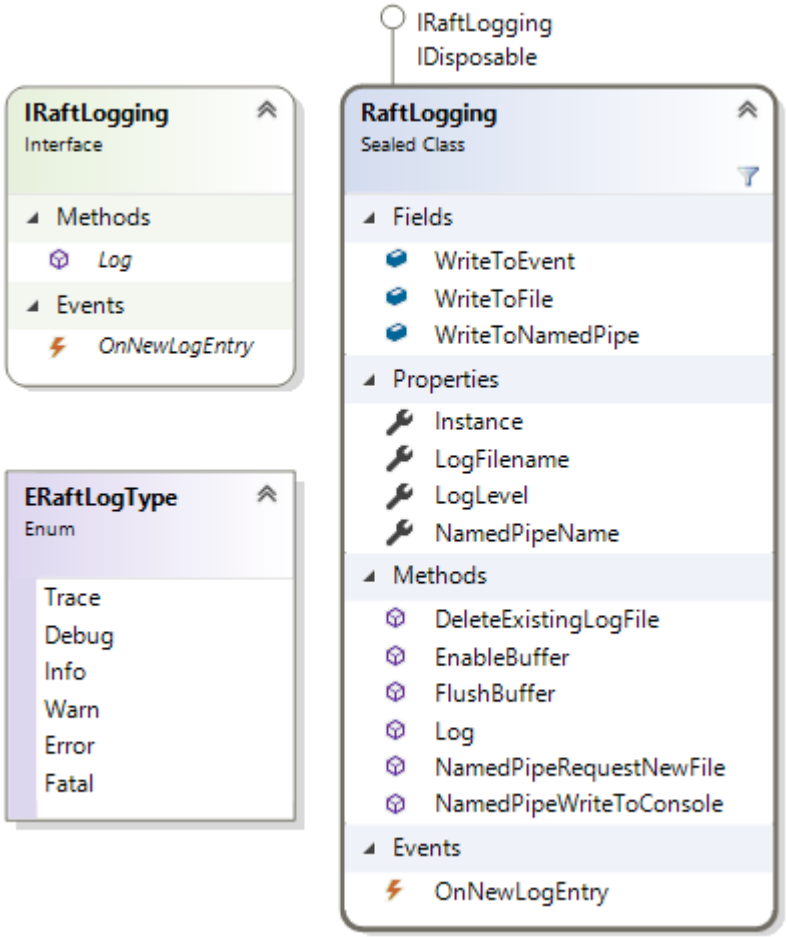

Figure 10 RaftLogging Class Diagram

```
private void SetupLogging()
{
    RaftLogging.Instance.LogLevel = raftConfigurationModel.LogLevel;
    RaftLogging.Instance.NamedPipeName = string.Format("RaftConsensus: {0}", NodeObject.Name);
    RaftLogging.Instance.WriteToNamedPipe = true;
    RaftLogging.Instance.WriteToEvent = true;
    RaftLogging.Instance.OnNewLogEntry += HandleInfoLogUpdate;
}
```

Figure 11 Excerpt of RaftLogging use within the Prototype application

Unit/integration testing

The RaftConsensus Library project uses the industry standard .NET unit testing library NUnit, which integrates with Visual Studio's Test Runner. Below you can see the class diagram covering all the various classes and helpers.

The setup of unit/integration tests is quiet standard, with the only perhaps non-obvious component being inheritance for various classes such as RaftConsensus, RaftDistributedLog, and UDPNetworking. This inheritance allows baseline tests to be run on all the different implementations. (e.g. in memory DistributedLog and persistent), as well allowing for further per implementation tests to also be declared as well for each implementation. For RaftConsensus, this inheritance is used to test all the various implementation options of the class, such as plaintext, encryption, number of nodes, number of active nodes.

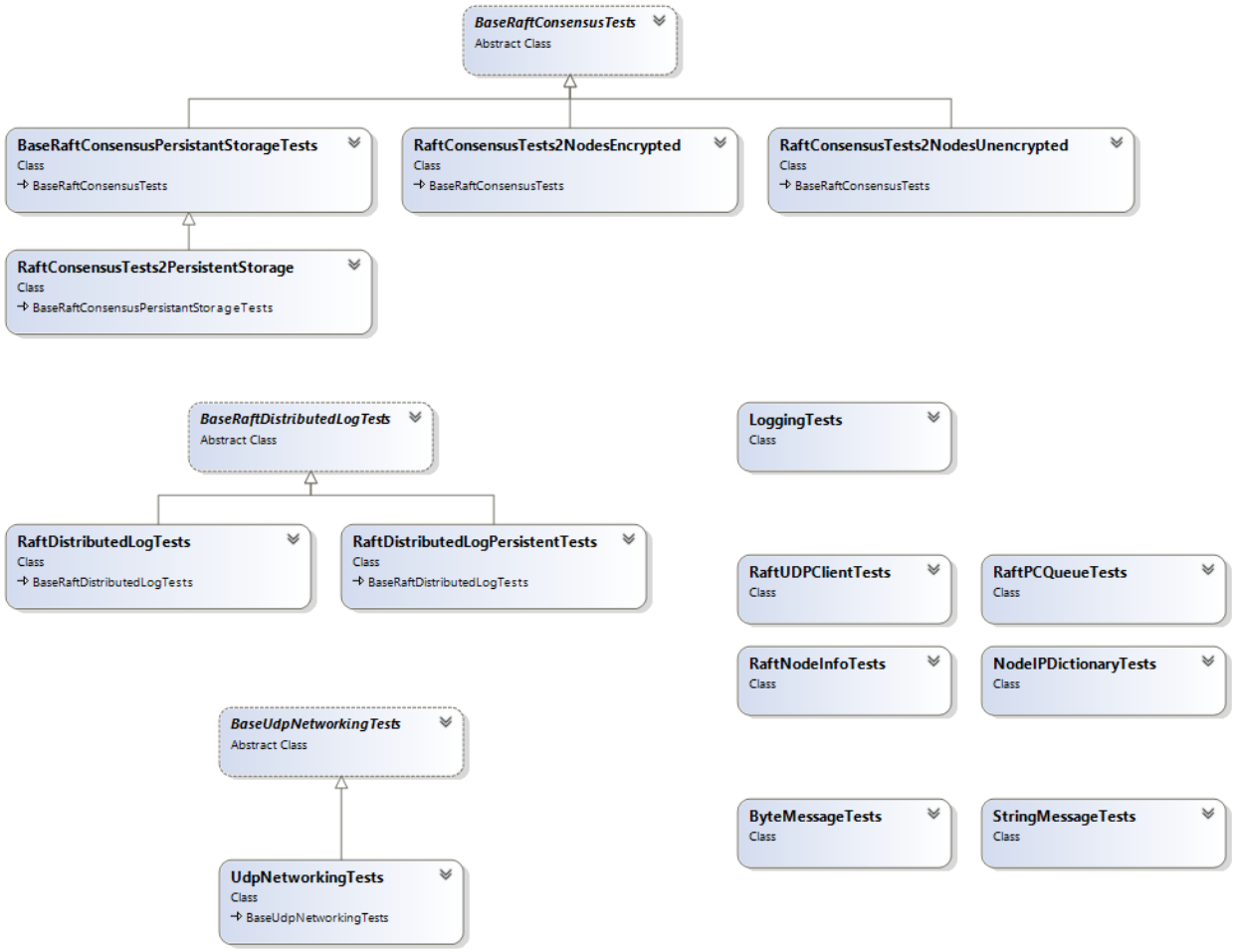

Figure 12 Test Suite Class Diagram

NuGet

NuGet is .NET's native package manager, and it's where this library is currently publicly available for simple integration into .NET applications. Eventually changes will need to be made to the RaftConsensus Library code base, and when that time comes Visual Studio (even community edition) can be used to produce a new NuGet version to be uploaded to the NuGet site by following basic publicly available steps. You will need to be added to the "Team" on NuGet which has package writing rights for the library. As our development so far has focused on removing any located bugs and implementing features before official release, we're currently hiding all but the latest library available from being offered by NuGet and only supporting the latest version.

Transition Phase Status Assessment

Results of Transition Phase Objectives

Contingency for the Completion of IOCM

Subject Coordinator gave a blanket extension to all projects for IOCM submission, telling to submit when they're ready. We had beta tests and manual information for persistent storage left to complete, so we did that and submitted.

Update Prototype Demonstrating New Features

Although it was in the library, we still had to add persistent storage into the prototype application. As this was going to get released publicly as our reference implementation code, we also spent the time refracting it into the cleaner industry standard MVC style code. This is discussed later in the Issues section below.

Publicly Release Application/Library

We successfully converted our private Bitbucket account to public for both the library and the prototype, so they're both now publicly available.

Contingency for Any Identified Issues

No further contingency was required during the T2 phase.

Produce user Documentation Library

User documentation was completed as part of the IOCM document, and developer documentation was completed as part of the PRM document. No extra documentation is required.

Resolve any Identified Issues

Some bugs were identified and resolved. These are further spoken about in the issues section below.

Begin Work on PRM

We successfully begun work on PRM on schedule and identified desired outcomes for the different sections of the assessment, as well as prioritize and estimate time required for each section. We brought forward with us the knowledge that clear diagramming will take a large amount of time, so we knew to focus on Programmer Documentation first.

Go Through PRM and Ensure Quality

After estimates of work required was completed, we got to work on the PRM assessment. It ended up taking us in the ballpark of 100 hours as expected. We've also gone and completed the extra step of going through the final document and ensuring we're happy with the quality of work produced.

Contingency for PRM

No contingency for PRM was required. As seen in the next section we successfully delivered PRM on time without extension needed.

Delivery of PRM

The completion of this document Phase Assessment, and it's merging into the final submittable document marks the successful completion of PRM.

Status for Project Risks and Mitigations

There a no more projects risks outstanding. All have been closed, or successfully resolved through mitigation strategies.

Issues Encountered

Needed to Add Persistent Storage Beta Test

We initially forgot to add persistent storage beta tests and manual entries into our IOCM submission, due to a blanket extension the subject coordinator gave out we successfully hurriedly added those into our assessment before resubmission.

Needing to Convert Prototype to MVC

The technical debt of the prototype because too large and unwieldy during this phase, and as the code is going to be released publicly as a reference implementation of our library it was important that it follows best practices by being in an MVC like structure. This was one of the biggest work items of this phase and was successfully completed within an iteration before we started PRM. Keeping us on schedule.

Demo Day Video

A bit of a curve ball was thrown to us when it was requested that we produce a 5-minute presentation on our project, our thoughts and our advice. This was an unseen requirement for developer time and took just over 10 hours to complete. However, we still managed to keep the project on schedule.

Bug with Displaying Persistent Stored Entries in Prototype

As expected this was a bug related to not refiring off the event of OnNewEntry when loading entries from persistent storage. This was resolved by providing extra functionality in the `IConsensus` interface to check for existing entries on load.

Needed Rebranding

Since we're coming up to a release of the project, we've realised we're actually using the official Raft .ico/png image everywhere without permission. Since our project isn't officially sanctioned, and we'd get shot down for asking for permission anyway, we needed to get new branding images for everything. We knocked up a few options and settled on the one shown in all of our public releases.

Found bug

This was found to be an accidental unsupported mixed configuration cluster, and as such did not need to be resolved.

Needing a large amount of time to complete the rest of PRM

Although PRM did take us the expected ballpark of 100 hours, we were able to successfully complete and submit it within the time frame budgeted.

Current Progress of Project

We have successfully completed this Transition Phase, and with it completed the ITC309 subject. We have achieved all required objectives for this phase, no project risks were left outstanding, and we resolved all leftover issues. We're very happy with the quality of work we've produced and the skills we've learnt during this project. And lastly, we are extremely confident and happy to consider our project status as successfully completed!