

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 – IOCM
Assessment 2

Joshua Cameron;Sean Matkovich

Contents

Implementation Model	3
Intended Production Environment.....	3
No Known Bugs.....	3
High Quality Code.....	4
Evidence of Best Practice Version Control	4
Feature Completion from Initial Project Aims.....	5
Results for Functional Requirements	5
Results for Non-Functional Requirements	7
Test Model.....	10
Master Test Plan Review - Stage Version 1.0 “beta ready”	10
Previous Test Result Revision	10
Version 1.0 Test Results.....	11
Test 25 - Full project code review	12
Test 26 - Redesigned unit testing suite for extended verification of existing functionality.....	13
Test 27 - Cluster can grow upon new node.....	14
Test 28 - Cluster can shrink upon losing node.....	15
Test 29 - UAS can attempt to change leader of cluster.....	16
Test 30 - Persistent log implementation (“Log Compaction”).....	17
Test 31 - (Optional feature) Support for upgrade path.....	19
Test 32 - (Optional feature) Completed performance analysis/optimization of code.....	20
User Acceptance Beta Tests	21
Beta Test 1 - Install prototype	22
Beta Test 2 - Create config - unencrypted/ephemeral storage.....	23
Beta Test 3 - Create config - network encryption enabled	24
Beta Test 4 - Create config - persistent storage enabled	25
Beta Test 5 - Create config - network encryption enabled/persistent storage enabled.....	26
Beta Test 6 - Join cluster - Join from creating config.....	27
Beta Test 7 - Join cluster - Join from loading config.....	28
Beta Test 8 - Append entry.....	29
Beta Test 9 - Receive commit entries	30
Beta Test 10 - Stop node	31
Beta Test 11 - Start node	32
Beta Test 12 - Survive node Failure	33
Beta Test 13 - Node re-join/rebuild distributed log.....	34
Beta Test 14 - Read developer logs	35
User Manual	36
Feature Phase Status Assessment.....	37

Results of Feature Phase Objectives	37
Focus on Code Quality/Refactoring.....	37
Redesign Unit Testing Suite	37
Dynamic Cluster Membership	37
(Optional) Detailed Performance Analysis	38
Persistent Storage of Log Entries.....	38
(Optional) Upgrade Path.....	38
Status for Project Risks and Mitigations.....	39
Issues Encountered.....	40
Re-evaluating Dynamic Cluster Membership	40
Debugging Refactored Code - Timing Issue.....	40
Encryption Bug Discovery	40
Contingency of IOCM Due to Holiday.....	40
Implementing SQLite	41
SQLite Library .dll.....	41
Current Progress of Project	41

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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	On track	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Setting on project

Mitigate project abandonment	7	Licensing allow for profit	<p>We've open sourced this project, so people are able to contribute or fork</p> <p>We're utilising the Apache2 license which allows for profit and business maintainers to continue to maintain code</p>	Success	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Reference thesis
Minimal resource usage	9	Consensus Log compaction	We've expanded on Raft Consensus's base implementation	Success	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	On track	<ul style="list-style-type: none"> • Testing coverage report • Link to High Quality Code above
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	<ul style="list-style-type: none"> • NuGet site link
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	<ul style="list-style-type: none"> • Link User Manual section of this document further down
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	On track	<ul style="list-style-type: none"> • Link to High Quality Code above

Performance	5	<ol style="list-style-type: none"> 1. Matches Paxos in performance of consensus 2. Own thread with ASYNC/non-blocking operations 3. Performance analysis 	<p>Raft Consensus Algorithm matches Paxos in performance, so no performance is lost</p> <p>Code has been written entirely multithreaded, with each node having at least 4 of its own dedicated threads, and there are no blocking operations.</p> <p>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</p>	Success	<ul style="list-style-type: none"> • Raft paper snip link • Link to iteration plan
Compatibility	6	<ol style="list-style-type: none"> 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 	<p>We've written this in .NET Standard to allow for cross platform development</p> <p>We've only got a popular cross platform JSON library, and Microsoft's own SQLite handler as dependencies</p>	Success	<ul style="list-style-type: none"> • .NET Standard 2.0 pic CDN link • NuGet dep. pic CDN link
Availability	7	<ol style="list-style-type: none"> 1. Can be run between servers locally or across Internet 	<p>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.</p>	Success	<ul style="list-style-type: none"> • Youtube link
Security	8	<ol style="list-style-type: none"> 1. Network level authentication 	<p>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</p>	Success	<ul style="list-style-type: none"> • Link to the secure networking file in BB

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	• Link to the secure networking file in BB
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

Test Model

Master Test Plan Review - Stage Version 1.0 “beta ready”

During our previous LCAM submission, we’ve outlined a list of 38 tests which directly relate to use cases, functional requirements, and non-functional requirements of the project. A successful completion of all those 38 tests would be directly related proof that all targets have been delivered.

Previous Test Result Revision

Of the previous 24 tests ensuring software functionality of which all were successfully passed, only one of those tests required revision from lessons learnt in the current phase. There are no previously failing tests which require reviewing and resolving.

ID	Feature/functionality	Testing environment	Acceptance criteria	Role	Planned Stage
5	Node authenticates using zero knowledge password proof	Dev. Evidence	Reasonable developer evidence provided	Joshua	Prototype

Although this was successfully implemented as the previous test result shows, it’s since had to be removed due to the discovery of a security flaw in the protocol used. Since no freely integratable libraries of the ideal SRP6A protocol were available at the time, the used protocol was written by hand through a public key exchange to verify identity. Although much thought was put into the protocol, there was an irreconcilable vulnerability in the initial protocol setup where anyone who captured the first message could man-in-the-middle (MITM) the rest of that point-to-point communication. All this code was immediately removed from the project for fear of “what’s worse than bad crypto? Crypto you don’t know is bad”. It all comes back to the “don’t write your own crypto” lesson, which we’ve learnt from this extends to crypto protocols. So, after some failed attempts at implementing SRP6A from the protocol docs themselves, the search begun again to locate open source licensed SRP6A code, which after multiple days of searching was eventually found. However, during implementation, it became clear that it was incompatible with our P2P architectures without major rework. SRP6A is built around server/client TCP communication, however we’re functioning in a P2P UDP style, and although around 20-30 hours were sunk into trying to get the code to function, it could not do so at the reliability level we’re aiming for with this project. We eventually fell back on simply using the SHA256 hash of the supplied password as a simple key derivation function and using it as the symmetric key for all inbound/outbound packets. The packets are encrypted with industry standard AES. Nodes are only able to communicate if they’re able to successfully decrypt the received packets, so this still provides security based on the strength of the guessable password however there is no key exchange protocol. Although we do plan to revisit this, a full SRP6A UDP/P2P implementation would prove as difficult as consensus itself, and as such falls outside the reasonable expectations of this subject. When the code is released publicly, we will ensure reasonable signage of the security used so developers can make their own informed decision.

Version 1.0 Test Results


Referring to the Master Test plan from the LCAM document, the following are the set of tests which we're expecting completion by this IOCM project milestone. We've called that milestone "version 1.0", which correlates with "beta ready, no known bugs".

ID	Feature/functionality	Testing environment	Acceptance criteria	Role	Planned Stage
25	Full project code review	Dev. Evidence	Reasonable developer evidence provided	Joshua	Version 1.0
26	Redesigned unit testing suite for extended verification of existing functionality	Dev. Evidence	Reasonable developer evidence provided	Sean	Version 1.0
27	Cluster can grow upon new node	Demo	Evidence of demo completing action	Joshua	Version 1.0
28	Cluster can shrink upon losing node	Demo	Evidence of demo completing action	Sean	Version 1.0
29	UAS can attempt to change leader of cluster	Demo	Evidence of demo completing action	Joshua	Version 1.0
30	Persistent Log implementation ("Log compaction")	Demo	Evidence of demo completing action	Sean	Version 1.0
31	(Optional feature) Support for upgrade path	Demo	Evidence of demo completing action	Joshua	Version 1.0
32	(Optional feature) Completed performance analysis/optimization of code	Dev. Evidence	Reasonable developer evidence provided	Sean	Version 1.0

Test 25 - Full project code review

Use Case		N/A - Full project code review					
Test Type		Developer Evidence					
Test Description		Full review of code base implementing a much cleaner approach to the underlying multi-threaded library					
Pre-Conditions:		Prototype library					
Post-Conditions:		Production ready library					
Notes:							
Results		Pass					
Step.	Step Description	Expected Result			Evidence Description	Evidence ID(s)	Result
1	Code refactor	Easier to read code base and logic. Simpler design. Added reliability. Resolve dealocking issues. Drastically reduce complexity of multithreading.			Relevant version control repository commits	1.1	Pass
Evidence ID		Evidence					
1.1		<div><div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div><div>?</div><div>Joshua and Sean</div><div>3aa9e50</div><div>Finished debugging the refactor of Consensus, works better than before and is simpler code!</div><div>2018-07-28</div></div><div><div><div>?</div><div>Joshua + Sean</div><div>c7a167b</div><div>Updated iteration plans and publish</div><div>2018-07-25</div></div><div><div><div>?</div><div>Joshua and Sean</div><div>b1095bc</div><div>Committing current refactoring work. Finished refactoring, just need to hunt down bugs now.</div><div>2018-07-21</div></div><div><div><div>?</div><div>Joshua + Sean</div><div>504b63b</div><div>Testing and code coverage.</div><div>2018-07-21</div></div><div><div><div>?</div><div>Joshua and Sean</div><div>65dc232</div><div>Updated README.md, changed the link for my name. And added link to the prototype.</div><div>2018-07-15</div></div><div><div><div>?</div><div>Joshua and Sean</div><div>4d32cc1</div><div>Committed work in progress. Refactoring the RaftConsensus class at the moment.</div><div>2018-07-15</div></div></div></div></div></div></div></div></div>					

Test 26 - Redesigned unit testing suite for extended verification of existing functionality

Use Case		Redesigned unit testing suite for extended verification of existing functionality			
Test Type		Developer Evidence			
Test Description		Review of code test suite			
Pre-Conditions:		Prototype library			
Post-Conditions:		Produced a unit testing library which supports using inheritance, and reaching a high level of code coverage			
Notes:					
Results		Pass			
Step.	Step Description	Expected Result	Evidence Description	Evidence ID(s)	Result
1	Code refactor	Test library supports inheritance, high level code coverage added	Relevant version control repository commits	1.1	Pass
Evidence ID		Evidence			
1.1		 <p>Joshua + Sean eb704cf More unit testing coverage currently at 88 percent. 2018-07-28</p>			

Test 27 - Cluster can grow upon new node

Use Case		Cluster can grow upon new node			
Test Type		Developer evidence			
Test Description		N/A			
Pre-Conditions:		N/A			
Post-Conditions:		N/A			
Notes:		This was part of the work item "Dynamic cluster membership". We've since made an evidence-based decision to not pursue implementing this feature due to the expected time requirements far exceeding time available.			
Results		No Action required			
Step.	Step Description	Expected Result	Evidence Description	Evidence ID(s)	Result
1					
Evidence ID		Evidence			

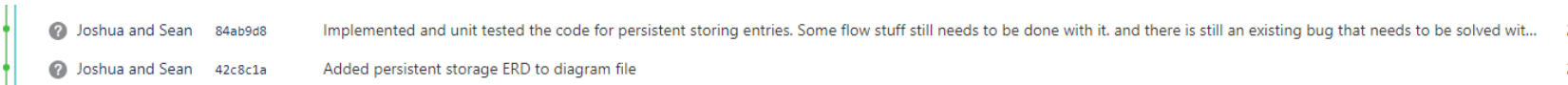


Test 28 - Cluster can shrink upon losing node

Use Case		Cluster can shrink upon losing node			
Test Type		Developer evidence			
Test Description		N/A			
Pre-Conditions:		N/A			
Post-Conditions:		N/A			
Notes:		This was part of the work item “Dynamic cluster membership”. We’ve since made an evidence-based decision to not pursue implementing this feature due to the expected time requirements far exceeding time available.			
Results		No Action required			
Step.	Step Description	Expected Result	Evidence Description	Evidence ID(s)	Result
1					
Evidence ID		Evidence			

Test 29 - UAS can attempt to change leader of cluster

Use Case		UAS can attempt to change leader of cluster			
Test Type		Developer evidence			
Test Description		N/A			
Pre-Conditions:		N/A			
Post-Conditions:		N/A			
Notes:		This was part of the work item "Dynamic cluster membership". We've since made an evidence-based decision to not pursue implementing this feature due to the expected time requirements far exceeding time available.			
Results		No Action required			
Step.	Step Description	Expected Result	Evidence Description	Evidence ID(s)	Result
1					
Evidence ID		Evidence			

Test 30 - Persistent log implementation ("Log Compaction")

Use Case		Persistent Log implementation ("Log compaction")			
Test Type		Developer Evidence			
Test Description		Unit and Integration test added log compaction feature			
Pre-Conditions:		Prototype library			
Post-Conditions:		User is able to persistently store their key-value store information to disk			
Notes:					
Results		Pass			
Step.	Step Description	Expected Result	Evidence Description	Evidence ID(s)	Result
1	Run test suite against "Persistent Log"	All test pass	Version control commits, unit tests results	1.1, 1.2	Pass
Evidence ID		Evidence			
1.1		 <p>  Joshua and Sean 84ab9d8 Implemented and unit tested the code for persistent storing entries. Some flow stuff still needs to be done with it. and there is still an existing bug that needs to be solved wit... </p> <p>  Joshua and Sean 42c8c1a Added persistent storage ERD to diagram file </p>			

1.2	<ul style="list-style-type: none"> ▲ ✓ TeamDecided.RaftConsensus.Tests.Consensus.Di... (40) 603 ms ▲ ✓ RaftDistributedLogPersistentTests (20) 603 ms <ul style="list-style-type: none"> ✓ IT_AppendManyNewKeyValuesToEmptyLog 453 ms ✓ IT_AppendManyValuesToEmptyLog 1 ms ✓ IT_AppendRequireingTruncate 5 ms ✓ IT_AppendToEmptyLog 1 ms ✓ IT_AppendUsingCorrectPrevIndexIncorrectPrevTerm 2 ms ✓ IT_AppendUsingCorrectPrevIndexPrevTerm 1 ms ✓ IT_CommitUpToIndex 2 ms ✓ IT_CommitUpToIndexTooLargeCommitIndexThrows 38 ms ✓ IT_ConfirmContainsKey 67 ms ✓ IT_GetEntryByKey 1 ms ✓ IT_GetEntryByKeyAfterManyEntries 2 ms ✓ IT_GetEntryByWrongKeyThrowsException 3 ms ✓ IT_GetEntryFromIndex 10 ms ✓ IT_GetEntryHistoryByKeyAfterManyEntries 2 ms ✓ IT_GetTermFromIndex 7 ms ✓ IT_GetValueByKey 1 ms ✓ IT_GetValueByKeyAfterManyEntries 1 ms ✓ IT_GetValueByWrongKeyThrowsException 2 ms ✓ IT_GetValueFromIndex 2 ms ✓ IT_GetValueHistoryByKeyAfterManyEntries 2 ms 	
-----	--	--

Test 31 - (Optional feature) Support for upgrade path

Use Case		(Optional feature) Support for upgrade path			
Test Type		Developer evidence			
Test Description		N/A			
Pre-Conditions:		N/A			
Post-Conditions:		N/A			
Notes:		This was part of the work item “Support for upgrade path”. We’ve since made an evidence-based decision to not pursue implementing this feature due to the expected time requirements far exceeding time available. Also, in this case there was not a sufficiently reasonable benefit for the feature to be added as well.			
Results		No Action required			
Step.	Step Description	Expected Result	Evidence Description	Evidence ID(s)	Result
1					
Evidence ID		Evidence			

Test 32 - (Optional feature) Completed performance analysis/optimization of code

Use Case		(Optional feature) Completed performance analysis/optimization of code			
Test Type		Developer Evidence			
Test Description		Developer provides evidence of performance analysis/optimization of code			
Pre-Conditions:		Code has some functionality which may be optimized			
Post-Conditions:		Code which performs some form of functionality faster or more efficiently			
Notes:					
Results		Pass			
Step.	Step Description	Expected Result	Evidence Description	Evidence ID(s)	Result
1	Developers perform reviews of code	Optimizations are found and implemented in the code	Link to the code file which contained the optimizations	1.1	
Evidence ID		Evidence			
1.1		Description of change: We found an optimization to our implementation of the protocol. Instead of the leader waiting for the next heartbeat to be sent out before giving new entries to nodes to bring them up to date, the leader will instead directly respond with the next entry upon receiving a commit reply. This brings the total round-trip time for an update from (heartbeat interval + (2 * round trip time)) to just (2 * round trip time).			

User Acceptance 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 is 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		This is a test to ensure the installer works and users can successfully install the demonstration application	
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			
Step.	Step Description	Expected Result	Result
1	Download Raft Prototype from here	User download .msi file	
2	Proceed through the standard steps of the installer. Accept any warning.	User installs software to computer.	

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			
Step.	Step Description	Expected Result	Result
1	Open Raft Prototype from the shortcut made on the desktop	Raft Prototype program opens on computer	
2	Select ‘Create cluster config’	Takes user to Create Cluster Config page	
3	Use the example configuration of: <ul style="list-style-type: none"> 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 50s	
4	Select to ‘Build’ the config, and select where to save the file	Config builds successfully Opens dialog window File is saved to targeted location	

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			
Step.	Step Description	Expected Result	Result
1	Open Raft Prototype from the shortcut made on the desktop	Raft Prototype program opens on computer	
2	Select ‘Create cluster config’	Takes user to Create Cluster Config page	
3	Use the example configuration of: <ul style="list-style-type: none"> 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	
4	Select to ‘Build’ the config, and select where to save the file	Config builds successfully Opens dialog window File is saved to targeted location	

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 is able to 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			
Step.	Step Description	Expected Result	Result
1	Open Raft Prototype from the shortcut made on the desktop	Raft Prototype program opens on computer	
2	Select ‘Create cluster config’	Takes user to Create Cluster Config page	
3	Use the example configuration of: <ul style="list-style-type: none"> 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 50s	
4	Select to ‘Build’ the config, and select where to save the file	Config builds successfully Opens dialog window File is saved to targeted location	

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			
Step.	Step Description	Expected Result	Result
1	Open Raft Prototype from the shortcut made on the desktop	Raft Prototype program opens on computer	
2	Select ‘Create cluster config’	Takes user to Create Cluster Config page	
3	Use the example configuration of: <ul style="list-style-type: none"> 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 50s	
4	Select to ‘Build’ the config, and select where to save the file	Config builds successfully Opens dialog window File is saved to targeted location	

Beta Test 6 - 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 one of the Create Cluster use cases and is sitting at a preloaded Start Node screen	
Post-Conditions:		User starts a node on their computer, user starts other cluster nodes, and is able to join a cluster	
Notes:		If the user takes too long starting the other nodes, they'll eventually re-prompts to search again	
Results			
Step.	Step Description	Expected Result	Result
1	Select Node1 as the node to start	Node1 starts up and searching for the cluster, state is "initialising"	
2	Complete "Join Cluster - Join from loading config" Use Case for Node2	Node2 start up and searches for the cluster, finds Node1, creates cluster, one of the nodes starts their UAS	
3	Complete "Join Cluster - Join from loading config" Use Case for Node3	Node3 starts up, searches for and joins the cluster.	

Beta Test 7 - 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 is able to load a node from a config file, and have that node join a cluster	
Pre-Conditions:		User has completed a Create Config use case and has a Raft Consensus Config “.rcc” file	
Post-Conditions:		User has been able to start enough nodes to create their described cluster	
Notes:		If the user takes too long starting the other nodes, they’ll eventually re-prompt to search again	
Results			
Step.	Step Description	Expected Result	Result
1	Open Raft Prototype shortcut on the Desktop	Program starts up and prompt user for what they’d like to do	
2	Select “Start Existing Node”, and select “.rcc” file	User is presented with dropdown list of possible nodes to start	
3	Select the applicable node to start	Node starts up and begins searching for cluster	
4	Repeat steps 1-3 for as many more nodes are needed for cluster	User starts the rest of the nodes needed	
5	Nodes will form and create cluster	One of the nodes will show their UAS is started	

Beta Test 8 - Append entry

Use Case		Append Entry	
Test Type		Unsupervised User Acceptance Test	
Test Description		This ensures that a user is able to 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. This is why we aren't including seeing it committed in this use case.	
Results			
Step.	Step Description	Expected Result	Result
1	Find the Node with their UAS Showing as Running	User can identify which node is running the UAS	
2	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	
3	Select Append	The values disappear, a request shoots off in the background to append this entry	

Beta Test 9 - Receive commit entries

Use Case		Receive Commit Entries	
Test Type		Unsupervised User Acceptance Test	
Test Description		This ensures that a user is able to 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 is able to observe the entry becoming committed across the cluster	
Notes:			
Results			
Step.	Step Description	Expected Result	Result
1	Look at all the open nodes and observe in their log that "Hello"/"World" has appeared	"Hello"/"World" appears in the nodes logs	

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			
Step.	Step Description	Expected Result	Result
1	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	

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			
Step.	Step Description	Expected Result	Result
1	Select "Start" on a node in the cluster what was previously Stopped	Node starts up, joins cluster and sets it status to "Running"	

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 is able to 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			
Step.	Step Description	Expected Result	Result
1	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	

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 into 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			
Step.	Step Description	Expected Result	Result
1	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	

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			
Step.	Step Description	Expected Result	Result
1	User picks any nodes and selects to move over to the Log tab	User is able to view and scroll through the back-end developer log of the consensus algorithm	
2	User 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 is able to view and scroll through the back-end developer log of the consensus algorithm, they can now see heartbeat and message flow/processing	

User Manual

[The user manual is available here](#) for the source code

[The user manual is available here](#) for the Prototype Demonstrator which users can use to complete the Beta UATs

Effectively targets intended audience - The target users of the manual are professional developers working on mission critical applications. These would be educated, code-literate individuals who firstly care if the library implements their required functionality, and secondly how difficult/many man-hours it will take to implement. As this project is released as an open source library, the standard is that the instructions for implementation are made directly available on the project's version control page.

Provides support for all business scenarios - The major focus of the user manual on the project page is ensuring instruction for each use case is provided. As the user will be testing these use cases through the use of download/installing a Prototype Demonstrator, this instruction is provided through step by step screenshots, and explanation/highlighting of relevant information. This format allows a quick understanding of the user interface for developers trying out the software library's functionality.

Amount of information presented - Developers are not just simple users, as well as understanding how to implement the code, it's important to give a brief intelligible overview of underlying consensus systems. So in addition to all the instructions which cover all use case scenarios, there is also a section which uses explanation of 3 "under the hood" images to give the developer basic understanding of the underlying functionality.

Failure scenarios - As this project is focused around being a fault tolerant algorithm, there are very few failure scenarios included in the use cases. The only noteworthy one would be not having started enough nodes to create a cluster, which is covered in the manual by showing the error popup that comes up when a node is asking to retry to find the cluster.

Logical structure of the user manual - To better facilitate clear instruction to the developers looking at the project, the manual has been written in a way to link related items together. The "Join Cluster" use case comes before the "Append Entry" use case for example, leading to an iterative build of complete understanding of the available functionality.

Supports beta level testing without further developer support - This manual was created to directly support the beta level test, so the developers have every confidence the manual will support users completing the beta level tests without further developer support.

Feature Phase Status Assessment

Results of Feature Phase Objectives

Focus on Code Quality/Refactoring

The top non-functional requirement of this project is reliability, and a big part of code reliability is a focus on code quality. This project has maintained its commitment to high quality code by taking the first full iteration of this session for a major refactor of the consensus code base. The refactor focused on three key points, reducing multithreading complexity/deadlocking/race condition issues, greatly reducing cyclomatic complexity of all code, and breaking down classes into single responsibility.

This process was very successful, with the biggest achievements being:

- Reducing the number of threads handling mostly all object in the Consensus class to only one, practically eliminating deadlocking opportunities
- Removal of all previous deadlock avoidance code
- Great reduction in the state change methods' cyclomatic complexity
- Simplifying message validation and processing
- Timing/timeout system from the Consensus class separated by itself

Debugging this refactor was an immense task which is certainly not being forgotten here; it will be discussed in the issues section later.

Redesign Unit Testing Suite

Another big part of reliability is testability; the ability to verify functionality. As such, part of the refactor included refactoring the unit testing suite, with some of the highlights being:

- Breaking out all setup functions of Nodes into methods to be reused by each test
- Setting up inheritance of test suites so multiple variations of classes could be easily tested (number of nodes, encryption on/off, etc.)
- An increase in code coverage to 88%, a percentage which is a healthy balance regarding wasting time writing tests for functionality we may change in the future

Dynamic Cluster Membership

The decision to not implement dynamic cluster membership is one we did not take lightly. The debate around it revolved around that it would be an unprovable/unverifiable extension to existing functionality which did not already exist in the Raft thesis. The author of Raft had used mathematical proof tools to validate his protocol; this meant any valid implementation could benefit from its provability. We simply do not have the time in this course to design and implement this complex feature correctly, and certainly not provable correctly as it should be. We instead took the opportunity to focus on continuing code quality work items, and spending time doing performance analysis. This reasoning was discussed in depth with our lecturer during oversight meetings and was agreed to be the most reasonable solution and that extending Raft would be best left to honours projects.

(Optional) Detailed Performance Analysis

Although an optional part of the project, we were quite passionate about finding some time to implement at least some basic optimisations into the code. The opportunity of not completing dynamic cluster gave us some time to do so. The most important performance change we made was to the rebuild times of out of date nodes, and speed at which new messages propagate and achieve consensus. We were able to achieve up to a theoretical 6 times faster times for bringing a node up to date, and up to a theoretical 6 times faster time for reaching consensus (conditions of 150ms heartbeats, and 15ms latency). This was achieved by not waiting until the next heartbeat to send out the next message but sending them out as soon as possible.

Persistent Storage of Log Entries

This feature was initially thought to be quite complex, however due to some initial brainstorming we'd come up with some simple ways to achieve it and believed it could be done relatively quickly. However, although our brainstorming ideas were correct on our way to achieve it there were many hours lost in fighting the unnecessary nonuniformity and complexity of SQLite libraries in the .NET Standard framework. We implemented Microsoft's SQLite implementation, however due to issues with it later we had to change to the SQLite team's own implementation. We ran into the same problems as with the Microsoft implementation, however these were eventually able to be overcome. This will be further discussed in the Issues section below, but the ability for nodes to persistently store log entries was successfully added on time.

(Optional) Upgrade Path

This optional feature was not added due to time constraints and the fact it provided little benefit to users who were not part of a small non-considered edge case of maintaining uptime during live node upgrades. This optional feature may be considered again in the future time allowing, however compared to other desired features/optimisations it's unlikely and considered low priority.

Status for Project Risks and Mitigations

Please note, we'll only be considering risks were open during this phase. Any previously closed issues which were reopened will also be discussed below.

Risk	Risk Map	Status	Notes
Scope creep inflates scope	Yellow	Closed	All developers have agreed on all release features, and this being the feature phase is now completed
Estimates for milestones are inaccurate	Red	Closed	There are not enough work item milestones left that this could be considered a reasonable enough risk to monitor, and our previous usage of our contingency plan for dedicated re-evaluation of time estimates has worked.
Member is unavailable	Yellow	Closed	We are close enough to completion of this project that each member is confident on their ability to complete it without the other if required
Member is lost	Yellow	Closed	We are close enough to completion of this project that each member is confident on their ability to complete it without the other if required
Code quality issues	Red	Closed	A major refactor has been completed on the code and any code not up to quality standard has been fixed or removed. We're currently at a state of consistent high-quality code, and upon completion of the feature phase we're not looking on any more major code changes.
Users have inaccurate expectations	Green	Closed	User expectation has been controlled through the user of an online accessible manual with the project
Poor software quality	Red	Closed	Similar to above, a major refactor has been completed on the code and any code not up to quality standard has been fixed or removed
Security too complex	Red	Closed	Security was too complex, far too complex. So instead of considering ephemeral key exchange as part of the project, we're simply implementing symmetric encryption with the pre-shared keys used in a basic key derivation function
Multithreading introduced high level of difficulty of troubleshooting	Red	Closed	It did. However, all multithreading issues have been resolved in the refactor, and no known bugs exist. We've also functionally removed the ability for deadlocking or race condition issues to occur.

Issues Encountered

Re-evaluating Dynamic Cluster Membership

Although this was discussed above, it is also an honorable mention here as it was an unexpected issue we did encounter. Only lightly touching on the above discussion, this was regarding due to the complexity of doing it right was the size of a project in and of itself, and due to that complexity and time constraints it was not completed. This decision was discussed with our lecturer during an Oversight meeting, and we spent the time instead focusing on other work in the project. We've learnt that in future project we should be more realistic in deciding milestones, and that we should provide at least vague time estimates of them to detect issues such as these. However, I'm not sure that would have dissuaded us due to our misunderstanding just how important reliability is to the project, but there is a lesson somewhere there nonetheless.

Debugging Refactored Code - Timing Issue

This was arguably the biggest issue we encountered in this whole phase. During the process of the major refactor we'd redesigned and integrated a new timing system for heartbeat/timeout event handling. The new system focused far more on high level flows, and basic cyclomatic complexity. However, when used in the Raft algorithm there was timing bug occurring which we could not track down. Previously we'd troubleshooted multithreading deadlock/race condition issues, so that included reading a log line by line, understanding what is going on at each stage and progressing it. Troubleshooting this issue instead was although simpler theoretically, but due to not being to do with deadlocking or multithreading, it did require a greater degree of trace level logging to isolate the issue. Our required logging was so verbose we were running into performance issues with the program running slow which caused the issue to not occur, something we tracked down to I/O wait performance related issues due to the verbosity of logging. After testing solutions like caching (didn't give us the unwritten logs when an issue occurred) and writing to a file in a RAM drive (still had I/O wait performance issues due to RAM drive drivers), we eventually conceded we're going to have to add the ability to offload debugging to a separate program which doesn't crash. We investigated, we found and implemented named pipes, a system which is used for two separate programs to talk to each other within an operating system. Using this we were able to achieve around unbelievable accuracy of around the nanosecond level in our logs, and we were eventually able to use those to resolve the issue.

Before implementing this timing code into Raft, it had been tested thoroughly in a separate application, however the issue found was in our implementation of the library into Raft, so unfortunately the only lesson we can take away is continue to plan for unforeseen issues.

Encryption Bug Discovery

This was covered in-depth in the Previous Test Result Revision section above; however, it is worth mentioning here as it was an issue that occurred during this phase. Finding a critical security bug during development was an issue which was not able to be planned for. The integration of SRP was much more difficult than it was understood to be and implementing key exchange protocols by hand universally lead to their own issues. In future it should have been implementation of a proven and available security protocol or bust. It's better to highlight a lack of network security so people can plan around it, rather than incorrectly attempting something.

Contingency of IOCM Due to Holiday

Worth mentioning was that one of the team members took their planned holiday, something we'd agreed on early in the project however failed to consider to highlight on our Project Plan, and were required to use our planned contingency time. This worked our fine for us due to the contingency time allocated, however a lesson can still be learnt.

Implementing SQLite

Implementing SQLite into a program is something that should be considered relatively easy for almost all developers. But due to the fragmented nature of SQLite and the .NET Standard ecosystem, this was found to be more difficult than it needed to be. There were undocumented omissions of features in the implementation which were planned to be used by the design, so a redesign was required. This took on the order of hours to do and implement, which it was expected to take less than an hour.

SQLite Library .dll

We had an issue where all of our projects were running with the new SQLite code enabling persistent storage, however when using the installer to make a prototype to confirm functionality it was throwing exceptions regarding missing DLL files. It was found that although SQLite DLL files were included within the NuGet package, they still needed to be integrated separately into the installer which wasn't detecting the dependency.

Current Progress of Project

We're extremely confident in considering our project status as very good. We've successfully achieved our goal of "Beta ready, no known bugs", and we're confident that beta testing will not turn up any major issues as all primary functionality has been verified. Looking ahead to the next phase, we'll be completing our beta tests, fixing any found issues and proceeding onto final product release.

We've successfully achieved our Feature Phase goals, and we're ready for the challenges of the next phase!