**Implementation Model**

for

Team Decided - Raft Consensus Library

## 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 it’s intended production environment. Our prototype is able to demonstrate all the features of our library, and it does so as it’s 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 seperate classes doing too much into their own classes, and thirdly to reduce cyclomatic complexity across the whole codebase. This effort was successful, and has lead 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 commiting to master
  + Each developer ensures tests pass before commiting 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 | On track | * 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 | On track | * 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 | * 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 | * 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 JetBrain’s Resharper Visual Studio Add In to ensure all code is written to it’s built in style guideline, as well as it’s other optimisations | On track | * Link to High Quality Code above |
| 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 pic CDN link * Nuget dep. pic CDN link |
| 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 | * 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 |