# 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 it’s 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 quiet 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 15 ms 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 quiet 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 preshared 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

### Reevaluating dynamic cluster membership

Although this was discussed above, it is also an honourable 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 focussing 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 occuring 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 seperate program which doesn’t crash. We investigated, we found and implemented named pipes, a system which is used for two seperate programs to talk to each other within an operating system. Using this we were able to achieve around unbelievable accuracy of around the nano second 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 seperate 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 occured 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 universaily 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 was 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!