admin

[Draw your reader in with an engaging abstract. It is typically a short summary of the document. When you’re ready to add your content, just click here and start typing.]

[Document title]

[Document subtitle]

Contents

[Project Vision 2](#_Toc511471469)

[Introduction 2](#_Toc511471470)

[Positioning 3](#_Toc511471471)

[Problem Statement 3](#_Toc511471472)

[Product Position Statement 3](#_Toc511471473)

[Stakeholder Descriptions 4](#_Toc511471474)

[Stakeholder Summary 4](#_Toc511471475)

[User Environment 5](#_Toc511471476)

[Example Scenario 1: The On-Prem Password Manager company 5](#_Toc511471477)

[Example Scenario 2: The database company 5](#_Toc511471478)

[Example Scenario 3: The Video Game company 5](#_Toc511471479)

[Product Overview 6](#_Toc511471480)

[Needs and Features (Functional requirements) 6](#_Toc511471481)

# Project Vision

## Introduction

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.

Consensus algorithm work by ensuring all nodes in a cluster are in agreeance on the current state of the service, and continues uptime of the service during node failure or network link failures between the nodes.

The vision for this project is to create and publish a fully featured implementation of a Consensus Algorithm as an open source code library which would allow for application developers to implement consensus/fault-tolerance into their services as easily as possible.

The benefit to an implementing company could be twofold; firstly, solving the problem of increasing uptime of a critical service, and secondly by taking advantage of consensus distributed consistent log feature they can maintain service uptime whilst offloading server computation to the end user devices, directly saving hosting costs for the company.

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

## Positioning

### Problem Statement

Consumer driven requirements for always available services has pushed companies to eliminate any downtime from their offerings. These downtimes impact user experience as well as trust in their platform.

When consumers have poor experiences, and lose trust in a service they may consider alternatives solutions for the needs. This impacts the growth of a company, as well as their ability to extract the financial benefits of scale from having a large user base.

Companies which minimise or eliminate any impact from downtime for their end users may better compete in the marketplace for their services.

### Product Position Statement

Our proposed project targets developers who are responsible for implementing services which require high uptime. This open source library aims to fill the market position of a consensus algorithm which focuses on ease of integration and which is written in the language the developer is most likely to be using.

The benefits to a company would include reducing overall product development time, reducing the total cost of developing a highly available service, and by reducing the need/costs for consensus specialists to implement the desired feature. Reducing these costs will also lower the barrier for entry for companies which previously could not have reasonably implemented this.

Driving down development overhead costs can not only improve a company’s competitiveness in their market and allow them to capture the cost of unnecessary expenditure, the reduction of development time can also reduce the financial risk that companies accept during their projects which implement this technical feature.

## Stakeholder Descriptions

### Stakeholder Summary

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Responsibility** |
| **Project Team** | Our team; functioning as developers, as well as project sponsors and management | * Ensuring system is maintainable * Ensuring market demand * Ensuring features are working as desired * Ensuring system is reasonably secure * Monitoring of project development progress Investing their time into development |
| **Team Leader/CTO** | The individual who is responsible for selecting and assessing technologies that may be beneficial for inclusion into their services | * Assess the amount of time and cost of implementation of this library as their feature * Confirmation of the benefits of the library matching the business requirements * Comparison of alternative solutions for their requirements * Proposing library as solution for their requirements to project sponsor * Managing the project of implementation |
| **Project Sponsor** | The individual who has requested the inclusion of the feature into their service, and provide the financing for implementation | * Assess the proposal by the Team Leader/CTO in the context of cost to benefit to the company and realization of benefits * Responsible to the business for the success of the project and reasonable use of expenditure * Responsible for sponsoring the resolution of the business case |
| **Developer - Consensus Specialist** | The individual in the Project Team whose responsibility is to learn | * Responsible for the intimate knowledge of the library * Responsible for technical practical knowledge to understand consensus algorithm and implementation * Liaising as the specialist consult inside the Project Team for practical implementation of the library into the service |
| **Developer** | The individuals responsible for the integration of the library into their service | * Understanding their existing service such that they can liaise with the Consensus Specialist to plan how to implement * Implementing the feature into their service |
| **End user** | The end users who is pressuring the company for highly available service | * Not applicable |

## User Environment

As the reliance on always available services grows, there becomes more companies and individuals who rely on their services having high uptime. These companies have many, sometimes thousands of users relying on the availability of their services to perform important roles in their lives.

Some examples of the users would be:

* Companies who offload server session to customers, such as multiplayer games.
* Companies offer As A Service (aaS) products needing high uptime
* Companies who want to add high reliability to their services
* Database programmers

### Example Scenario 1: The On-Prem Password Manager company

This company produces a Linux webserver that their business clients run on premises, this server is a Password Manager for all their employee’s company logins details and supplier logins in. Their biggest customer, a 1000 seat enterprise, has complained of the risk of reliability in this password web server and would like to mitigate the risk of failure of the server as it’s stop hundreds of people from conducting their daily duties. They suggested that they were interested in a more reliable solution which mirrors itself between sites to maintain its uptime, is transaction safe, and importantly never loses data or returns incorrect results.

### Example Scenario 2: The database company

Consensus algorithms are complicated, and databases are a critical part of almost all web services. In order to bring consensus to the masses, a database company is looking to implement clustering at the transaction log level of their database, so that web servers can failover to other nodes in the cluster if one falls over. The job of developers greatly simplified as they only need to write heartbeat checks to their database, and upon failure select another node in the cluster.

### Example Scenario 3: The Video Game company

There is a video game company who runs a popular multiplayer turn based strategy mobile game, many thousands of players are playing at any one time. Analysing their company costs, hosting costs accounts for around 40% of yearly expenditure, which is second only to staff costs. Each game server hosts up to 10 players at a time, meaning the company needs scale up to run hundreds and potentially thousands of these servers during peak playing times. The idea came up about offloading running the server onto one of the players playing in the server, however it was shot down due to the ability for players to leave at any moment causing bad experiences for the other players in the servers. They’ve now had the bright idea to employ a consensus algorithm which synchronises server state among their players, so if someone leaves the game server starts running on another player. This will dramatically cut hosting costs, while not impacting uptime for the players.

## Product Overview

### Needs and Features (Functional requirements)

Priority scale is between 1 and 12, with 1 being the most important

|  |  |  |  |
| --- | --- | --- | --- |
| **Need** | **Priority** | **Features** | **Planned Release** |
| Consensus between distributed systems | **1** | Replicated log, with consensus algorithm | Proto. |
| Fault tolerant distributed service | **2** | Consensus algorithm allows for a fault tolerant distributed system | Proto. |
| Improved reliability of existing service | **3** | System is fault tolerance, so it will improve reliability | Proto. |
| Complete proven reliability | **4** | Based on proven algorithm | Proto. |
| Minimal additional surface area for failure | **5** | Complete coverage unit testing | Version 1.0 |
| Cross Platform | **6** | Targeting .NET standard framework | Proto. |
| Mitigate project abandonment | **7** | Licensing allow for profit | Proto. |
| Minimal overhead/impact to service performance | **8** | Equivalent to leading consensus algorithm, Paxos in performance | Proto. |
| Minimal resource usage | **9** | Consensus Log compaction | Proto. |
| Ability to pick ideal leader | **10** | Fitness considered in selection of next leader (new leader calls for election of ideal leader) | Version 1.0 |
| Warm nodes | **11** | Tracks log but doesn’t vote | Version 1.0 |
| Upgrade path | **12** | Versioning built in, backwards compatibility minor releases and single major | Final |

### Other Product Requirements (Non-functional requirements)

Priority scale is between 1 and 14, with 1 being the most important

|  |  |  |  |
| --- | --- | --- | --- |
| **Requirements** | **Solution** | **Priority** | **Planned Release** |
| Reliability | 1. Full coverage unit testing | **1** | 1. Final |
| Usability | 1. Designed to be as simple as possible to integrate. 2. Released as Nuget package | **2** | 1. Proto. 2. Final |
| Documentation | 1. Full coverage documentation for algorithm and API | **3** | 1. Final |
| Quality | 1. Full coverage unit testing 2. Strict adherence to style guide | **4** | 1. Final 2. Proto. |
| Performance | 1. Matches Paxos in performance of consensus 2. Own thread with ASYNC/non-blocking operations 3. Performance analysis | **5** | 1. Proto. 2. Proto. 3. Version 1.0 |
| Compatibility | 1. Written in .NET the second most popular language. 2. Minimal dependencies. 3. Written in .NET standard, cross platform 4. Agnostic networking option 5. Designed to be as simple as possible to port languages | **6** | 1. Proto. 2. Proto. 3. Proto. 4. Version 1.0 5. Proto. |
| Availability | 1. Can be run between servers locally or across Internet | **7** | 1. Proto. |
| Security | 1. Network level authentication | **8** | 1. Proto. |
| Privacy | 1. Security measures to join cluster | **9** | 1. Proto. |
| Scalability | 1. Dynamic cluster membership, horizontal scaling | **10** | 1. Version 1.0 |
| Testability | 1. Open source code, unit tests provided | **11** | 1. Final |
| Extendability | 1. Open source code | **12** | 1. Final |
| Auditability | 1. Open source code 2. Logging | **13** | 1. Final 2. Proto. |
| Troubleshooting | 1. Verbose logging | **14** | 1. Proto. |

### Business Justification for Functional and Non-functional Requirements

First and foremost, the most important feature of this project is increasing the reliability of a User Application Service by maintaining a replicated log; hence the top 5 functional requirements link directly to this. Not only does the desired system require to spread a replicated log amongst nodes, it also needs to be fault tolerant in the case of failure of a minority of nodes, and manage the start/stop of a UAS in cases where only one instance should run. An unconventional part of reliability is usability, therefore there is a focus on the design of the usability of the library to ideally eliminate issues to reliability caused during implementation. To reinforce usability for the product, there will also be complete documentation available.

Companies looking to run this software may look to do so on any operating systems (e.g. Windows, Linux and mobile), so it’s important that the code is written in a way that allows for this portability. To implement this, the project will be written in the .NET Standard Framework, which allows portability over all platforms .NET is able to run on.

Part of the benefit of releasing all of the code for this project as open source is that the business risk of project abandonment by ourselves could be picked up by the client, or even a potential open source community. This would enable security updates, bug fixes, and new features to continually be added to the project; and the licensing the project is released under allows for companies to profit from the code.

A fact of implementing a consensus under the hood of a service is that there will always be some overhead; the consensus algorithm being implemented has the same performance as the more popular/complicated and error prone Paxos algorithm. So, using Raft gives us all of the benefits, without the understandability/usability downsides of Paxos. This minimal performance overhead allows companies implementing the library to not suffer from unreasonable latency overhead while waiting for servers to reach consensus, which would’ve caused a direct impact to their user experience. With careful design and the right conditions, it’s even be possible to for this library to be used in a real time 30-60 tick game.

When only a single instance of the User Application Server (UAS) is required, such as a video game server, it may be important to the user experience that most ideal node in terms of latency, or hardware performance runs the UAS. Having the ability for the nodes to pick the most ideal leader for consensus based on performance enable this improved quality of service.

When a node fails or leaves the cluster, this impacts the redundancy/reliability of the cluster as a whole. To bring back a safe state, an additional node is brought in to bring the cluster back up to the desired number of nodes. However, if the cluster service has been running for a long time the replicated log may be so large that it takes several minutes to get another node ready. To resolve this issue, a requirement for allowing Warm nodes to be updated semi-regularly with committed entries to save time later is important. This directly improves reliability and user experience upon additional node failures/departures.

As companies require always on services, an important part is that the consensus library also include backwards compatibility, so allow for rolling software updates through the nodes in a cluster. This allows for the updating of UAS without causing service downtime to the service, avoiding any customer impact.

Security and privacy are also both highly important since these nodes will connect across the network, in a modern-day service there is an expectation of a duty of care to users for the protection of their potentially sensitive information or data.

Part of the easy of code maintainability is facilitating the recreation of any possible bugs in the library; as such, to enable the troubleshooting of this the library will output at multiple debugging levels based on desired configuration.

Implementing scalability in the cluster by enabling the dynamic adding and removing of nodes allows the integration of this library with the horizontal scalability necessary for modern infrastructure design.

#### Justification of ordering of requirements

The library being quite atomic in nature (i.e. it either works or it doesn’t when ready for integration) has impacted the ordering of the requirements priorities. They have primarily been focused on functionality first before the extra “nice to haves”, so that the product can get out the door as fast as possible, while never compromising on the reliability. For example, Security is absolutely necessary in a production release of this library as it may be conducted over the Internet (and other publicly known vulnerabilities are exploited in similar software, discussed elsewhere in this document), however it’s technically not required to have a running consensus algorithm.

# Requirement Model

## Use Case Model

After investigation of the requirements the following [Use Case diagram](https://media.discordapp.net/attachments/418342556800385048/433932325508743168/2_-_Use_Case_Diagram.png?width=720&height=585) has been developed to outlining the initial set of interactions required

|  |
| --- |
| Use case diagram  Figure 1 |

## Short Use Case Descriptions - In Table Form

### Actor Key

UAS = User Application Server  
Dev = Developer

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Use Case Name** | **Precondition** | **Actor** | **Need** | **Do Something** | **Basic Intent/Goal** |
| Join Cluster | Has information about the cluster | UAS | Join the cluster | Join Cluster | Become a member of the cluster |
| Run Cluster | Join Cluster use case has run | UAS | Participate in cluster actions | Run Cluster | Start participating in the cluster |
| Receive Committed Entries | N/A | UAS | Bring itself up to date | Receive Committed Entries | Can maintain consensus |
| Append Entry | N/A | UAS | To add a message into consensus | Append Entry | Message gets committed into the consensus |
| Receive Notice of Entry Commit | N/A | UAS | To know if the message they requested to be committed, was committed | Receive Notice of Entry Commit | So that they may update the running UAS state |
| Receive Start Server | The current node should be running a UAS | UAS | To be running a UAS when required | Receive Start Server | They may go ahead and start up their UAS to provide services |
| Receive Stop Server | The current node should stop running their UAS | UAS | To stop running a UAS when required | Receive Stop Server | They stop acting as a UAS and providing service |
| Respond with Server Fitness | UAS implementing own fitness function | UAS | Wants to respond with its fitness for being the UAS | Respond with Server Fitness | The most fit node may be identified to run the UAS |
| Read Entry Value | N/A | UAS | Read log entries that have been committed | Read Entry Value | Can reference the committed data |
| Send Network Message | Offloading node communication to UAS | UAS | Send a message offloaded to it from the underlying consensus algorithm | Send Network Message | The underlying consensus algorithm can still communicate without relying on it’s own networking stack |
| Receive Network Message | Offloading node communication to UAS | UAS | Forward a message offloaded onto it’s networking stack to the underlying consensus algorithm | Receive Network Message | The underlying consensus algorithm can still communicate without relying on it’s own networking stack |
| Confirm Identity | N/A | UAS | Confirm the identity a new node communicating for the first time | Confirm Identity | Establish security through trust, and ensure they cannot be man in the middled |
| Read Logs | N/A | Dev | Understand what the underlying consensus algorithm is doing, perhaps tracking a bug | Read Logs | Understand what the underlying consensus algorithm is doing |

## Domain model

|  |
| --- |
| Figure 2 [Domain Model](https://cdn.discordapp.com/attachments/418342556800385048/433932360279654410/2_-_Domain_Model.png) |

### Discussion regarding domain model

We’ve found this is easiest to explain using an example of a distributed multiplayer game running a which is in a server/client setup. In this example the server is also running on one of the clients. In that example, the UAC would be the game client which the players would be interacting with, and UAS would be the game server all the game clients are talking back to. Our library would be integrated into the UAS.

Reading from the top left of the diagram it’s shown how the UAC (i.e. game clients) talk to the active UAS (i.e. game server) in the Cluster Application Servers(CAS). Then it’s shown how the CAS is made up of UASs. It’s the library which links these UACs together into a cluster, and the UACs simply talk to whomever is the ‘leader’ UAS in the CAS. The UAC can simple just try each of the IPs in the CAS until they find the leader who responds, this cuts down on the complexity in the UAC software.

Next, the UAS uses the Consensus API to talk to its internal Consensus Node. This consensus node is the part that handles cluster operations, and part of that is instructing the UAS to let it know when it should and shouldn’t be running the active game server. The nodes all communicate to each other through messages which are sent and encrypted by the Networking Library. Consensus between the nodes is through a shared distributed log (Distributed Log) which the active UAS commits entries into, and the follower nodes are updated with. Each node stores information about the other nodes which are in the cluster with it. And each node does verbose logging for troubleshooting reasons.

### Does the analysis of non-functional requirements demonstrate an ability to identify, quantify, prioritise, and communicate required system qualities? (LO2)

This is the part of the document where the above would be analysed, however after discussing with Jim, he has confirmed that this is covered by our ”Business Justification for Functional and Non-functional Requirements” section in the Project Vision above.

# Architecture Proposal

## Purpose

This document describes the philosophy, decisions, constraints, justifications, significant elements, and any other overarching aspects of the system that shape the design and implementation.

## Architectural Goals and Philosophy

The most important goal of this project is increasing the reliability of a User Application Service; hence all the high priority non-functional requirements link directly to this and it’s also the primary architectural design goal. By directly designing for reliability we incorporate it foundationally into our final product; part of this reliability focused design is a large amount of time initially invested into accurately and exhaustively planning out the project to meet all the functional and non-functional requirements.

An underlying requirement of reliability at the code level is rigorous full suite testing and a focus of strong adherence to code quality/style guidelines. This reduces errors, as well as improves maintainability.

Key to the implementation of reliability is usability, therefore there is also a focus on the design of the usability of the library. To reinforce usability for the product, there will also be complete documentation available to ease integration.

The final library needs to maintain functionality in a system with up to a minority of its nodes failed and will be put into potentially hostile environments in terms up frequently failing nodes.

## Assumptions and Dependencies

Assumptions:

* Market need for consensus library, specifically in .NET
* Performance and latency overhead of consensus is manageable without negatively impacting customer experience
* Requirement for security and the impact of that overhead is acceptable
* The limitation of compatibility in cross-platform implementation for mobile is limited to Xamarin is acceptable
* Dynamic cluster membership, including increase and decreases cluster size/scalability is an important feature to the developers
* Designing this library as free and open source software will not detrimentally inhibit company adoption and that auditability is important
* That developers would utilise the debug logging features of the library to diagnose their issues
* That developers would report back any bugs or issues with library back so they may be resolved

Dependencies:

* Developer implementing the programming required to read/write User Application Service state information into the consensus log
* Developer is able to start a User Application Service off of solely the information contained within the consensus log
* Developer implementing or having load balancing logic in any User Application Client accessing the active User Application Service. This may simply include failover at the IP level between the Cluster of Application Servers.
* Developer will either manually specify node networking addresses or enable a lobby system in their own User Application Service to supply them as required to the library for the starting of the cluster

## Architecturally Significant Requirements

* The system must be able to communicate over a network
* The system must be able to maintain consensus between distributed network nodes
* The system must be able to maintain a consistent replicated log
* The system must be able to survive failures up to a minority of it’s nodes
* The system must be as easy as possible to implement into existing projects
* The system must be as reliable as possible

## Decisions, Constraints, and Justifications

|  |  |
| --- | --- |
| **Decision** | **Justification** |
| C# language using .NET framework | C# is the [second most popular language of 2017](https://insights.stackoverflow.com/survey/2017#most-popular-technologies), so using it will make our library available to a wide range of developers |
| C# over Java | The more mature library integration framework of C# (Nuget) over Java outweighs the 5 percentage points of difference in their popularity |
| .NET Standard | .NET Standard is a subset of functions in the .NET framework which is. Although this makes development more difficult due to the reduction of prebuilt libraries |
| Bitbucket | Developer familiarity and the ability to have repos private during development |
| Full suite unit testing | As reliability is the most important functional requirement, we want to ensure not only that the code is working as intended, but also that changes to the code impact functionality |
| Style guide | Part of releasing open source code while easing development and maintenance of it by disparate developers is ensuring strict adherence to coding standards |
| Code review | With reliability such an important non-functional requirement, systematic independent party examination of code is put in place to find bugs as early as possible, as well as maintain adherence to style guide. |
| Documentation | To enhance usability, thorough cleary and consistent documentation |
| Implementing own network library | The existing functionality did not exist in an available library, or existing libraries were excessively bloated. With networking foundation to the node communication, controlling it was a practical decision. |
| Test driven development | As we’ve focused on exhaustive design for reliability reasons, TDD is a complementary fit due to its ability to ensure that all of the designed requirements are implemented |
| Visual Studio IDE | Industry standard .NET IDE, developer familiarity, ease of unit testing |
| Security | Always-on security was decided due to the nodes being designed to cluster across the public internet, as well as avoiding common issue of leakage or exploits such as the recent [etcd](http://www.zdnet.com/article/thousands-of-etcd-server-installs-are-leaking-credentials/) and [memcached](http://www.zdnet.com/article/memcached-ddos-the-biggest-baddest-denial-of-service-attacker-yet/). |

|  |  |
| --- | --- |
| **Constraint** | **Justification** |
| Consistent public API | Maintain ease of implementation for developers, regardless of underlying implementation changes |
| Minimalistic approach | When deciding on additional features or functionality, careful consideration must be taken to ensure a minimalist design is adhered to such as that the implementation of the software maintains being as simple as possible. This is due to reliability reasons, by aiming to avoid potential issues caused during implementation by over complicated or extensive options for developers. |
| Usability first approach | When deciding on approaches which the developers interacts with (such as the API), the most important constraint is usability and must be considered over unessential functionality. This usability first approach increases reliability, which has been justified multiple times above. |
| UDP | Although the guaranteed ordering and acknowledgment of packets in TCP would be ideal for a consensus algorithm, the additional round trip times required for this increase latency overhead unreasonability, and as the consensus algorithm takes cases of consistency and packet loss, UDP was chosen as the network protocol so we did not need to incur this overhead. |
| Universally standard data structure for distributed log | A key-value store (also known as “dictionary” in .NET) was chosen for it’s speed, reliability and non-complex implementation for developers. |

## Architectural Mechanisms

### Architectural Mechanism 1 - Distributed Consistent Log

This is the component used by the User Application Service (UAS) 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 start up in the event of a running UAS failure.

### Architectural Mechanism 2 - Fault Tolerance

This is the features which allows an increase in availability of a given service, it does this though enabling the failing over of a User Application Service to another available node in the cluster.

### Architectural Mechanism 3 - Network Communication

This is the functionality which allows the distributed consensus nodes to communicate with each other. It will based on the “fire-and-forget”/ ”connectionless” UDP protocol to reduce latency, while leaving the overhead of handling packet loss to the consensus algorithm.

### Architectural Mechanism 4 - Security

This feature provides identification through public key cryptography, confidentiality through encryption, and integrity through HMAC, for all network communications.

### Architectural Mechanism 5 - API Based Integration

The User Application Server (UAS) communicates with it’s consensus node through the use of a .NET class library. This single interface focused on usability is how the UAS communicates to the consensus algorithm.

### Architectural Mechanism 6 - High Quality Code

There will be 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 UAS high availability, it’s focus on quality must be paramount.

## Layers or Architectural Framework

From the below diagram, it’s shown where each Architectural Mechanism exists in the design. These architectural mechanisms in the diagram are numbered to reflect the assigned numbers previously in Architectural Mechanisms above.

|  |
| --- |
| Figure 3 |

#### Where each NFRs is addressed in the architectural framework

|  |  |
| --- | --- |
| **NFR** | **Related Architectural Framework Mechanism** |
| Reliability | Fault tolerance |
| Usability | API Integration |
| Documentation | API Integration, however handled externally to the system |
| Quality | High Quality Code |
| Performance | N/A, this is handled at the algorithm and code level |
| Compatibility | N/A, this is handled by choice of code language |
| Availability | Networking |
| Scalability | Distributed Consistent Log, handled in consensus algorithm |
| Security | Networking |
| Privacy | Networking |
| Testability | High Quality Code |
| Extendability | High Quality Code |
| Auditability | High Quality Code |
| Troubleshooting | High Quality Code |

## Architectural Views

### Use case view

Please see Use Case Diagram

### Logical view

Please see Domain Model

### Physical view

The consensus library will be integrated into the User Application Service, and will be used to logically tie multiple UASs together into a Cluster of Application Services (CAS).

In the example case of a distributed network where UASs are offloaded from centralised company servers onto UACs there is only a logical difference between UAC and UAS, not physical.

|  |
| --- |
| Figure 4 |

In the example case of dedicated linked servers which provide services for clients, there are multiple distinct servers which run the UASs forming a Cluster of Application Services (CAS). The User Application Clients communicate with the CAS through simple IP failover style.

|  |
| --- |
| Figure 5 |

# Technical Competency Demonstrator

[Link to the Technical Competency Demonstrator](https://bitbucket.org/teamdecided/technicalcompetencydemonstrator/src/master/)

[Link to the networking code library (RaftNetworking)](https://bitbucket.org/teamdecided/raftnetworking/src/master/)

After analysing the project, the most fundamental technical process which we identified is the sending of messages between nodes. All functionality of distributed consensus is built upon the ability to send/receive messages from nodes in a timely fashion. After our initial findings of there not being a publically available non-bloated networking library which suited our needs, we determined we needed to write our own. This is what we’ll be doing for our Technical Competency Demonstration (TCD).

This decision was not taken lightly, as writing socket level programming in any language is fraught with error/difficulty and technical programming traps for new players. Fortunately, one of our team members has prior experience writing libraries like this, in C, C++, and the C# language as well.

When estimating performance bottlenecks in the raft consensus protocol running across distributed nodes, the largest factor which impacts consensus time of the network is the node’s network latency from one another. Therefore, we chose UDP based networking over TCP, this was due to UDPs lack over acknowledgement message overhead for each message reducing overall message latency. The risk of running UDP however is that as the packets are connectionless, messages may be lost after transmission without the TCP resolving them in its host-to-host OSI layer (layer 4). This has been deemed not an issue for our project, as the raft consensus algorithm itself handles internally any packet/message loss issue.

Initially we thought that we’d be caught out with a maximum packet size issues, as with UDP networking over the internet you are only minimally guaranteed 576 byte messages due to historic networking device buffer sizes, and a likely modern packet size in the 1472 byte range. At such small packet sizes, we’d need to implement our own packet reassembly protocol into the messages to support our larger message requirements, however upon testing we found that Microsoft’s socket wrapper UdpClient was already implementing this and allowed us to send messages with a total length of 65KB, much more than we’ll reasonably require.

During one of our oversight meetings with Jim, he confirmed that our team’s technical competency demonstration would have to have the following attributes:

* Serialisation/deserialisation of a message in JSON
* Sending the serialised message over the network, and deserializing it to write the output
* Single threaded

We decided to go above and beyond these specifications and implemented a multithreaded networking library consisting of 3 synchronized threads. We realised it wasn’t worth writing code for a TCD to just throw it away after, and that we could take this time to write useful code now and then we’d be able to use it later on in the project.

## Class diagram

|  |
| --- |
| Figure 6 Class Model |

As you can see from the class diagram, the networking class itself is a single class which is interfaced with through its minimal public methods based on an Interface, this class sends/receives messages derived from the BaseMessage class through the property of polymorphism. The library minimally only needs to send ByteMessage messages, however we’ve added StringMessage messages as well to simplify our example TCD program. In practise in the consensus library we’ll be deriving our own messages based on our needs of message types.

## Example TCD Program

|  |
| --- |
| Figure 7 |

## Example TCD Program output

|  |
| --- |
| Figure 8 |

For usability/demonstration reasons, the one example program is used for displaying the two networking classes communicate. The TCD example program references the dynamic linked library (DLL) file compiled in the RaftNetworking project. It works by completing the following steps:

* Instantiation the UDPNetworking classes so they may be used for communication
* Registering an OnReceivedMessage functions for handling received message processing
* Starting the UDPNetworking classes listening on the designated ports, this starts all their multiple threads
* Manually registering the IP details of each client with one another so they’re able to communicate
* Then there is a loop of sending 100 messages both ways between the two classes. When the OnRecievedMessage function is called, you can see from its console output the out-of-order properties of UDP messages and how they’re also occurring asynchronously.
* Then we simply sleep to allow the previous asynchronous actions to perform before writing out our “Waiting to stop…” message, this is so the message displays at the bottom with its readline function
* After the user presses enter on their keyboard, the program then stops the networking classes, which stop their threads, and that completes the program

Network level Wireshark packet reassembly confirmation

|  |
| --- |
| Figure 9 |

The last image is of actual networking wire output captured through Wireshark, where it can be seen Microsoft’s socket wrapper does indeed break up a message of 65KB into 1514 byte packets with 1480 byte payloads. The target IP address had to be a non-localhost host address, as localhost traffic does not pass through the networking stack where Wireshark can read it.

Although this library is fully functional for our needs as a TCD, before implementing it into our own project we’ve highlighted additional features we’d like to add:

* Smoothly handling all exceptions with detailed error messages
* Adding full unit testing to ensure code quality
* Perform code review and ensure the code meets style guidelines
* Add a logging ability, likely with the NLog library
* Implement a heartbeat mechanism
* Ensure all functions are thread safe
* Derive a class which implements our desired security protocols
* Clean-up various code smells, closing of listening thread and isNetworkingRunning flag