Flock: A Peer-to-Peer Low Latency Distributed Storage Middleware for the Internet of Things

Giovani Luigi Rubenich Brondani[1], Ana Paula Canal[1]

[1]Computer Science Undergraduate Course
Universidade Franciscana (UFN), Santa Maria, RS – Brazil

giovaniluigi@hotmail.com, apc@ufn.edu.br

Abstract. *This paper describes the implementation of a peer-to-peer middleware for distributed data storage. The focus of this paper is on the code architecture and on the data models used to achieve eventual data consistency. As a result of this work, it was produced the code that implements the middleware, along with unit tests to verify the correct implementation in C# language, using the .NET Standard 2.0. The middleware is intended to be used in various scenarios, from servers to embedded devices, where eventual consistency is acceptable.*

Resumo. *Este trabalho descreve a implementação de um middleware peer-to-peer para armazenamento distribuído de dados. Este artigo foca na arquitetura do código e nos modelos de dados usados para atingir uma consistência eventual dos dados. Como resultado desse trabalho foi produzido o código que implementa o núcleo do middleware, junto com testes unitários para verificar a correta implementação em linguagem C#, usando .NET Standard 2.0. O middleware é destinado à utilização em diversos cenários, de servidores a dispositivos embarcados, onde consistência eventual é aceitável.*

1. Introduction

In recent years, a major technology shift has been happening. The ever-growing availability of internet connectivity is enabling several objects of our world to be connected to a computer network, creating the Internet of Things (IoT).

Objects which were previously unconnected, that are all around us, are being provided with the ability to communicate with other objects and people. For Hanes et al. (2017, p. 32): "IoT focuses on connecting 'things', such as objects and machines, to a computer network, such as the Internet".

Most of the technologies used on the Internet rely upon the client-server model, which centralizes data, and into a few protocols. With this new world of interconnected objects, different needs arise.

> "The IoT devices generate a mountain of data […] when all this data is combined, it can become difficult to manage and analyze it effectively. Therefore, IoT systems are designed to stagger data consumption throughout the architecture, both to filter and reduce unnecessary data going upstream and to provide the fastest possible response to devices when necessary." *[Hanes et al. 2017, p. 48]*

Devices participating in IoT networks, often exchange a massive amount of data usually originating from sensors. And to make the situation worse, most of the time those devices operate in a bandwidth-constrained network, and/or in a power-constrained environment. Sometimes IoT devices even have mobility.

There are several different physical network solutions employed in IoT. However, many of those solutions might not be able to deliver the network structure, addressing, and throughput required by most middleware used on the Internet.

1.1. Proposal

This paper presents Flock, a peer-to-peer distributed storage middleware, with eventually consistent data. The middleware is intended to be used on top of different types of physical networks, including low throughput networks, like those often used for communication between devices of the Internet of Things (IoT).

It is also important to recognize that connectivity is never a guarantee, even in highly connected parts of the world. The only way to achieve a seamless experience is by adopting solutions that are operational even when the network is not available [Offline First 2021].

Flock's eventual consistency is a relaxed consistency model that is often adopted by large-scale distributed systems where availability must be maintained, despite communication disruption, and where delayed consistency is acceptable [Vogels 2009].

Flock establishes an overlay network[1], where nodes of identical responsibility can share data using a set of pre-defined data models. According to Elmasri and Navathe (2011), a data model is a collection of concepts that can be used to describe the data types, relationships, constraints, and operations that are possible to be applied to the data.

A multi-model[2] approach is employed to cover different demands. The best data model can be selected by the middleware's client application, depending on the properties and behavior of the data to be exchanged. For example, a stream of immutable data, such as a log of events over time, can be stored in a data set that only grows over time and accept only insertion operations. Numeric data structures, that are updated often, can be kept in a structure that prioritizes speed and offers robust and fast concurrency control for frequent updates. Different data types will experience different system performances and behavior.

Flock is also planned for scenarios with high mobility of the nodes, such as drones (formally known as Unmanned Aerial Vehicles, or UAV) since according to Genc et al. (2017), drones are an emerging form of new IoT devices.

---

[1] An overlay network is a computer network that is layered on top of another network [Tarkoma 2010].
[2] Term used to designate a database that uses multiple data models [Pimentel 2015].

1.2. Main Objective

The main objective of this paper is to describe the concepts and to implement the middleware's main portion of code, using interfaces or abstract types where a system part is desired to be modular, i.e., open for replacement according to different needs or different platforms.

1.3. Specific objectives

- Define the middleware internal architecture, allowing for flexibility and support of different storage and communication systems.
- Implement data models suitable for usage in distributed data systems considering the lack of coordination and high concurrency.
- Design a communication model, to be used in the overlay network, that: is partition tolerant, introduces a small overhead, and supports cryptography.

2. Theoretical background

This section introduces the main concepts required for the understanding of this paper.

2.1. Internet of Things

A definition of the Internet of Things is given by Oracle (2021a) which states that the IoT: "describes the network of physical objects – 'things' – that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the internet".

The basic premise of the Internet of Things (IoT) is to join objects that currently are not connected to a computer network. The goal is to make it possible for these objects to communicate and interact with people and other objects, enabling tighter integration between the physical world and computers [Hanes et al. 2017].

To connect resource-constrained devices, such as embedded devices, extensively used on the Internet of Things, dedicated data storage and management system is often required. Low bandwidth, limited processing power, and restricted memory are great limitations for embedded devices joining a regular computer network. Therefore, it seems important to have access to alternative solutions, such as the one proposed in this work.

2.2. Distributed Systems

A distributed system can be thought of as a set of independent entities that cooperate to solve a problem. From a hive of bees to a flock of birds, cooperative systems exist since the beginning of the universe.

Regarding the definition for distributed systems in the context of this work, Bapat (1994) says: "A distributed system is an information-processing system that contains a

number of independent computers that cooperate with one another over a communications network in order to achieve a specific objective".

Based on the definitions above, it is possible to observe that Flock is a system with distributed characteristics, and therefore it can be viewed as a distributed system.

2.3. Middleware

The main objective of this work is to develop a distributed storage system that simplifies the development of applications that need to share data through a peer-to-peer network. It should be especially relevant where there are frequent changes to the network's structure.

By using the concept of middleware, the solution can be structured to be decoupled from the client software. A definition from Oracle (2021b) says: "Middleware is the software that connects software components or enterprise applications. Middleware is the software layer that lies between the operating system and the applications on each side of a distributed computer network".

For Etzkorn (2017), middleware is a layer to hide the lower-level complexity of networks from the application programmer. It allows for a client running in one machine, to communicate to another machine, independent of their operating system and hardware.

Flock can be used as a building block for other systems, freeing the developer from the concerns of the underlying communication and storage system.

2.4. Peer-to-Peer Systems

The overlay network required for this work needs to be de-centralized and non-structured, where the nodes can communicate directly to each other. Such a concept is known as a peer-to-peer system and has existed for a few decades now.

Peer-to-peer (P2P) computing as described by Vu, Lupu, and Ooi (2010) has some desirable aspects, such as the symmetric role for the nodes, heterogeneity regarding the nodes' hardware and capacity, and dynamism of the connection.

The fundamental characteristic of a P2P system, which is what makes it appropriate for this work, according to Steinmetz and Wehrle (2005) is that it consists of autonomous entities (peers) in a decentralized network, geared toward the shared usage of distributed resources.

2.5. Conflict-free Replicated Data Types

A CRDT is a Conflict-free Replicated Data Type. According to Almeida, Shoker, and Baquero (2016), it is a distributed data structure that automatically resolves conflicts in a way that is consistent across all replicas of the data. In other words, the distributed data

is guaranteed to eventually converge globally, as in an eventually consistent distributed system.

The CRDT was formally defined by Shapiro et al (2011) and is presented as: “a replicated data type for which some simple mathematical properties guarantee eventual consistency”.

The CRDT is employed where there is a need for a data structure to be replicated across multiple nodes in a computer network. It is used where replicas need to be updated independently and concurrently without coordination between the nodes. These data types offer a mathematical guarantee to resolve inconsistencies that might occur.

Different CRDTs emerge as viable solutions to handle the problem of concurrency found in the scenarios of distributed systems, where there is no coordination, such as proposed by Flock.

The CRDTs, unlike other eventual consistency approaches, may strongly simplify the development of distributed applications [Bartolomeu 2015].

2.6. Software Technologies

For code development, the main technology used is the .NET Standard, version 2.0, which according to Microsoft .NET (2021) is a specification of APIs available in all the different framework implementations of the .NET.

The chosen programming language is C# since it is the main supported language by the .NET ecosystem. The C# compiler is available for different platforms and was originally designed to work together with the .NET Framework.

2.7. Test-Driven Development

Due to the complexity of distributed systems, the development cycle of such a system can be quite challenging. It seems adequate to apply a methodology that ensures that each implemented feature is completely correct, before proceeding to other features.

Test-Driven Development (TDD) is a practice that consists of creating unit test cases. It is an iterative approach that combines programming, the creation of unit tests[3], and refactoring.

Since the proposed middleware architecture extensively uses interfaces, that can have different concrete implementations, this work is a good candidate for applying test-driven methodologies, which are based on mocking parts of the system under test.

According to Bender and McWherter (2011), Test-Driven Development is different from many other development methodologies, where the developer starts by

[3] A unit test is a test designed to test one requirement for one method [Bender and McWherter 2011].

creating the classes and the code. With TDD the developer starts by writing tests that will fail. The first failure, that will prevent the tests to compile, is going to be the missing classes. Once those classes are created, the tests will still fail, because the class and methods that were just created still do not do anything. So, the next step is to write code inside the methods to make the test pass. Enough tests should be added, to ensure that all requirements of the feature being tested are being met. When the entire requirement has been expressed in tests, and all the tests pass, the development is complete.

3. Related Work

There are many distributed storage solutions available in the literature. In this section, a few systems, similar to the one proposed by this paper, will be presented.

The XMIDDLE middleware proposed by Mascolo et al. (2001) focuses on the application of distributed systems for mobile hosts. It presents a system capable of handling network disturbances due to mobility, such as low bandwidth and loss of connectivity. In the XMIDDLE's network, nodes are structured as a tree. This structure is reasonable only when the number of nodes is relatively low since the load over nodes closer to the root becomes to grow when the node count grows.

OrbitDB is a serverless, distributed, peer-to-peer database. It uses a distributed File System as the underlying storage system. It is therefore a layer of abstraction on top of a distributed file system. OrbitDB also uses CRDTs for data synchronization and therefore it is a database suitable to be used in many different scenarios [OrbitDB 2021].

OrbitDB uses a single CRDT along with content-based data addressing (using data's content hash as a unique identifier) as the base for all its data models. However, the project works towards the creation of a decentralized web and is not meant to be a solution for communication between IoT devices.

Riak is a NoSQL high-performance distributed database, designed to deliver high availability and to be scalable. The system is available in different versions, each one optimized for a particular data model. For example, Riak KV has a key-value design and delivers different data models for storing massive amounts of unstructured data. Riak TS is optimized for time series data, and it can handle massive amounts of data and has been optimized for IoT [Riak 2021].

It is an excellent option for building IoT networks where the nodes can interact with a centralized database server. However, this system is a centralized database, and it was not designed to be installed in IoT devices, but just in the server that receives the data from those devices.

The middleware proposed in this work is built on top of the robustness and reliability of the CRDT theory. Unlike the other work mentioned, it offers a complete communication system through a peer-to-peer overlay network. Flock is cross-platform, has low latency, and is intended for usage in IoT devices.

4. Methodology

For the implementation of the middleware, it is proposed the usage of a methodology based on Test-Driven Development including additional steps to decompose a high-level block diagram, into empty classes. These empty classes, then are used to apply the TDD practice.

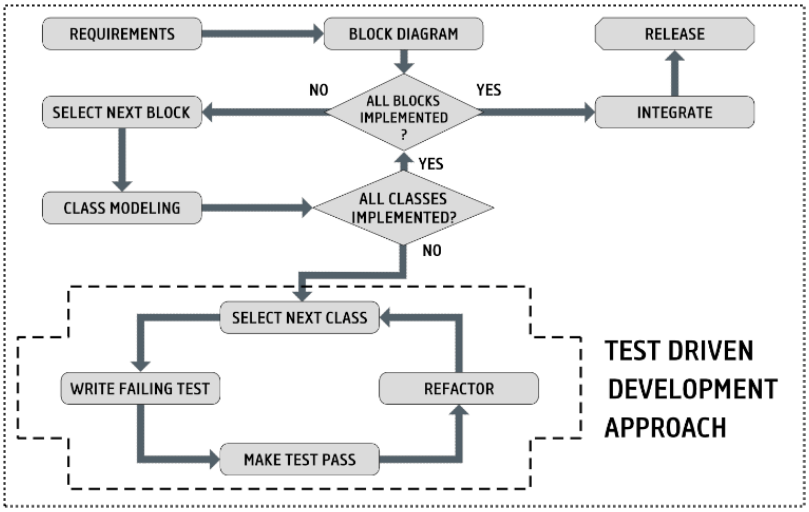

Figure 1: Methodology diagram

In this considered methodology, the process starts with the high-level requirements of the system as shown in Figure 1. The system is described in terms of its main functionalities and capabilities. For this work, the following requirements are presented:

- The middleware's architecture should be designed around the concept of exchangeable code modules, for different implementation.
- Network nodes should communicate through a peer-to-peer network, where they have identical roles, forming a uniform network.
- The entire system should be tolerant to network partitioning.
- The main communication protocol should have a reduced overhead.
- Data access should occur through a key-value store like API, where the keys are globally unique identifiers.
- The API should support several data models, offering different options for the distinct applications' problem domains.
- Data synchronization must ensure conflict resolution and data convergence for all supported data models.
- Data storage and network communication should be open for cryptography support to be implemented in the future.

Based on the project requirements, the high-level block diagram from Figure 2 was constructed. With the block diagram, the development starts by selecting one of the functional blocks.

The selected block is decomposed and modeled through classes from the objected-oriented paradigm supported by the C# programming language. Once the class model is available, the classes are declared in the programming language, using empty methods that initially will throw exceptions.

For each class, a set of unit tests is built to test all expected functionality of the class under development. The class then has its methods populated with logic to provide the results expected by the unit tests. Once all tests have passed for a given class, the next class of the currently selected functional block is chosen, and the process then repeats.

When all functional blocks have their classes implemented, the blocks are then integrated, and the development is finished.

5. System Design

The diagram in Figure 2, illustrates the proposed functional blocks of the system, and their containing .NET assembly[4]. This paper focuses on the development of the "core" assembly. As shown in the diagram, the "core" assembly contains the code responsible for managing the framework, controlling data replication, resolving data conflicts, and controlling the communication between the nodes.

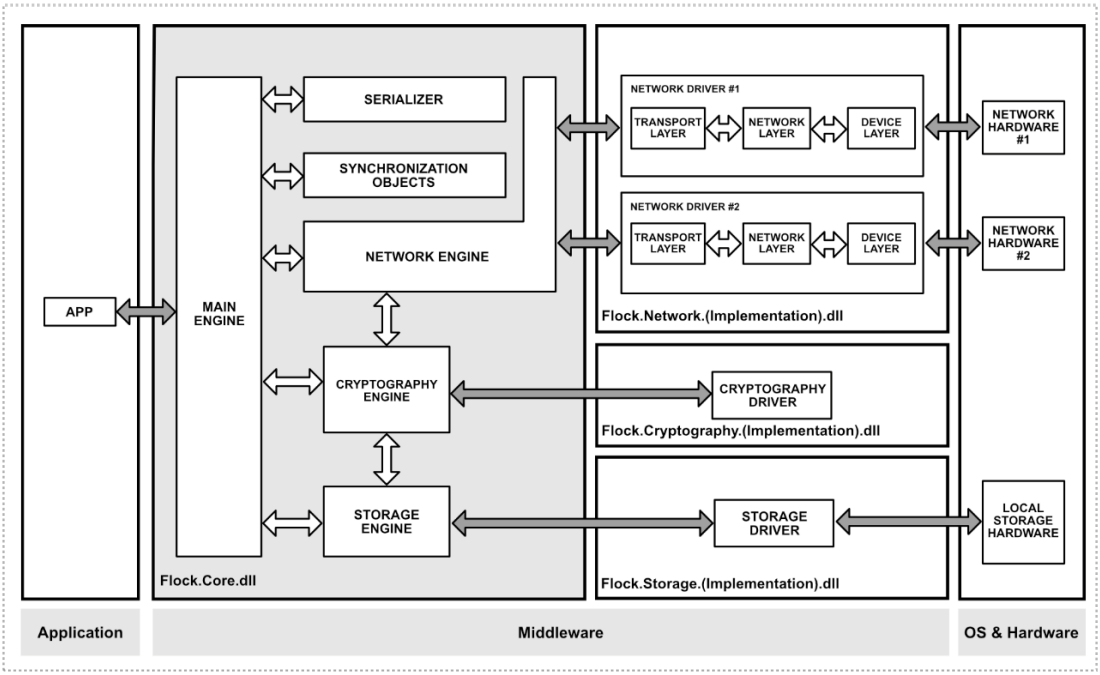

Figure 2: Block diagram

Individual assemblies are used to isolate dependencies when implementing platform-specific code, such as storage drivers and the network stack for each of the supported communication systems. A device that implements a storage driver that, for

[4] Assemblies are the fundamental units of deployment in .NET-based application and take the form of executable (.exe) or dynamic link library (.dll) files [Microsoft Docs 2019].

example, uses MySQL database does not need to share its dependency on MySQL's specific binaries, with another device whose storage driver uses, for example, SQLite database.

The separation in different assemblies also allows for easy extension and changes in the middleware. Different platform support, cryptography mechanisms, and encoding/decoding algorithms can be added to the system, without requiring the core logic to be modified, and not even re-compiled.

5.1. Functional Blocks

Through the diagram in Figure 2, it is possible to see the overall architecture of the middleware. This section presents a detailed view of each functional block.

5.1.1 Main Engine, Synchronization Objects, and Serializer

The "Main Engine" is responsible for the middleware management and client API. It represents a controller that coordinates the subsystems and maintains the entire middleware working. This block handles the data replicas and resolves data conflict, with the support of the "Synchronization Objects" and the "Serializer" blocks. It also exposes a public API, which can be used by the client applications, to interact with the middleware.

5.1.2 Storage Engine and Storage Driver

Flock middleware is not tied to a specific storage system. The storage engine provides a bridge between the platform-specific implementation of a storage system and the rest of the middleware.

Through the "Storage Engine" functional block, the other blocks can store and query data from local storage, without dealing with specific details of different storage systems, which then are implemented in a separated block named "Storage Driver".

Different storage drivers may be used to support, for example, different database management systems which will persist data for the middleware.

5.1.3 Cryptography Engine and Cryptography Driver

Cryptography is essential in any network communication. In the proposed middleware, data encryption can be used both for network communication and local storage.

The "Cryptography Engine" functional block offers an abstraction level to the other blocks, acting as a bridge between the "core" assembly and any implementation, while a "Cryptography Driver" implements specific algorithms to secure data. Different algorithms have different performances and require different platform resources.

The exact algorithm, however, is implemented by the "Cryptography Driver" which exists in a .NET assembly outside the "core" assembly and therefore its implementation is not within the scope of this paper. For this work, suffices to know that the communication between two nodes is assumed to be confidential, and only intelligible by those two nodes. This requirement must be fulfilled by all implementations of the "Cryptography Driver".

5.1.4 Network Engine and Network Drivers

The "Network Engine", as its name suggests, is a functional block that manages the communication between nodes. It abstracts the middleware internal logic from the details of implementation-specific "Network Drivers".

Flock middleware can work through different physical networks at the same time. The network implementation is provided by a separated .NET assembly as in Figure 2. Each network implementation requires a specific "stack" of layers, inspired by the OSI Model[5]. It is up to each network driver implementation to handle protocols for packaging, segmentation, as well as error check, and several other responsibilities.

When the middleware is used over computer networks employing the Internet protocol stack, for example, these layers are not required to be implemented when the operating systems already implement them. Instead, only a simple class can offer a bridge between the operating system resources and the middleware's network driver interface defined in the "core" assembly.

5.2. Data Models and Synchronization

Data between nodes in Flock's network is accessed through a key-value store like API. The keys are unique identifiers, while the values are CRDTs implemented for each data model.

Each node participating in the network has its own globally unique identifier (GUID) with a size of 128-bit. The unique identifier can be derived by applying a hash function over unique identifiers present in the hardware, or it can be generated by a stochastic algorithm, usually a pseudorandom number generator.

All data entries are spread to all nodes, via direct replication and gossip-based[6] dissemination. The nature of CRDTs makes it possible to perform updates from any node without coordination. Concurrent updates from different nodes will automatically be resolved by the merge function, which all data types must provide, and which should be monotonic so that the data never suffers a rollback.

---

[5] The OSI Model is a layered reference model for network communication developed by the International Standards Organization (ISO) [Tanenbaum and Wetherall 2011].

[6] Gossip-based dissemination is a protocol for broadcast. A survey on gossip protocols can be found in Leitão et al. (2010).

The middleware should be eventually consistent and dedicated to providing high read and write availability (partition tolerance), with low latency. Note that in an eventually consistent system, such as Flock, a read may return an out-of-date value.

5.2.1 Implemented Data Models

Different applications call for different data models and thus different CRDT are used for these models. The significant differences between different CRDTs are in how the data is stored or represented, how data of multiple nodes are merged and how data gets output when reading.

Flock uses the CRDTs as a building block to build higher-level generic data models, which are then easily adapted to application-specific needs. The data models to be initially supported by Flock are:

- Counter: Represents a signed 128-bit decimal value, without rounding error, with support for increment and decrement operations.
- Feed: Uses a CRDT to represent a set of items in time, where items can be added or removed from the collection. But once removed, an item cannot be added back.
- Blob: It is a data model where blocks of bytes can be treated as a single piece of information. This type of data is useful to represent complex binary structures that are immutable. It uses a CRDT for conflict resolution based on versioning.
- Map: Works like a dictionary of items, where each item is a value accessed by a unique key. The items can be added and removed. Once removed an item cannot be added back. Each item also supports updates. When an item is updated, it uses a CRDT for conflict resolution based on versioning.

5.2.2 Data Access and Synchronization

All the data access and synchronization are controlled by the “Main Engine” functional block. The “Main Engine” will take the application data, represented through a data model, and will replicate it between the nodes in the network using CRDTs to handle synchronization.

The CRDTs structures handle data consistency and synchronization. They are decoupled from the other pieces of code and require synchronization between the different nodes.

The middleware goes through a synchronization phase during data access and other events, such as when network data is received, to reduce the entropy[7] in the system. During the synchronization phase, the middleware will send data from one node to the other nodes. The data model implementation will inspect the received data and decide

[7] In the context of distributed databases, entropy means data replicas going out of synchronization [Mukil 2020].

how it should merge the local data with the received data. When all nodes in the network have a CRDT replica with the same state, the CRDT has converged.

5.3. Class Modeling

An overview of the class diagram created after the block diagram decomposition is shown in Figure 3. Several interfaces are defined in this assembly to allow for different implementations, as discussed earlier in this paper. Concrete implementations for the interfaces are injected into the middleware when an instance of the "FlockMiddleware" class is created.

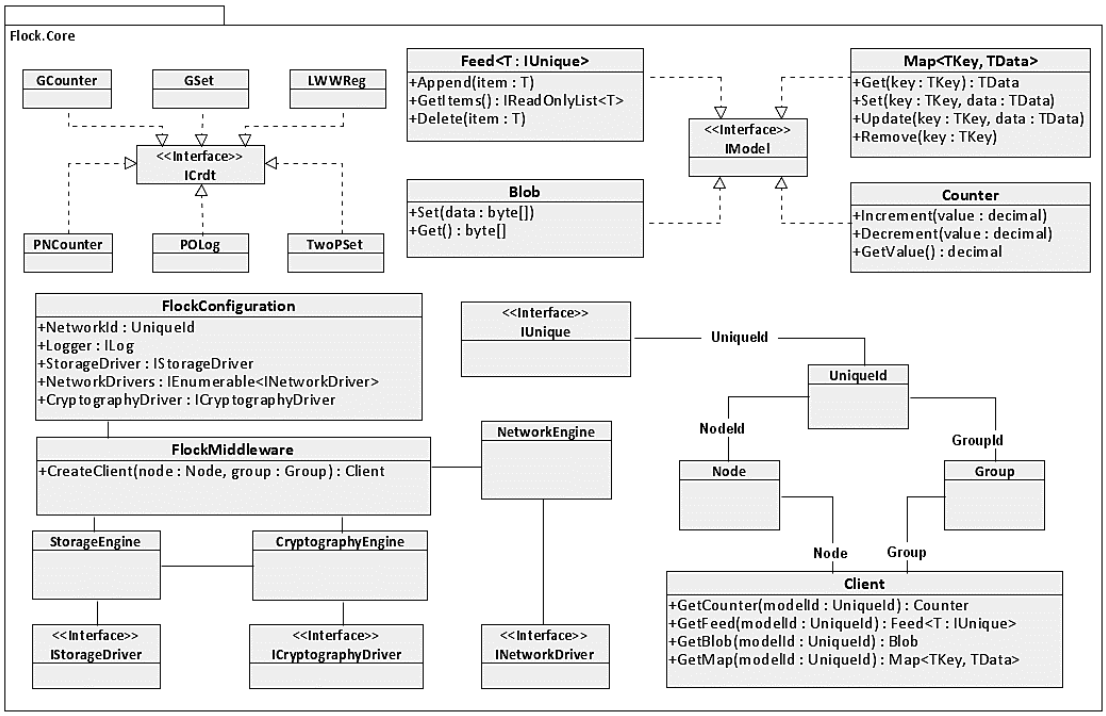

Figure 3: Simplified class diagram for the Core assembly

The class "Client", is created by the method "CreateClient()" from a "FlockMiddleware" instance and works as the main API that the user will utilize for interacting with the middleware. The methods in the class "Client" work as a key-value store for data models, as per the requirement expressed in Section 4.

6. Test Results and Preliminary Review

After the implementation of the core of the middleware, it becomes possible to test the entire system by injecting instances of drivers for tests. Figure 4 - Unit test to detect if a counter increment generates a network packet shows a unit test that performs a complete test of the middleware using an instance of a "Counter" data model.

Inside the unit test, the middleware is set up using a test configuration object ("FlockConfiguration") containing drivers implemented for test purposes only. A node object is created to represent the local node. A client object is created, which exposes

methods to acquire data model objects. A “Counter” is requested with a new “UniqueId” generated. The “Counter” is incremented, and the unit test verifies that after this increment the network driver received a packet. The test checks if the packet received by the driver contains the same “UniqueId” as the one used when acquiring the “Counter” object.

Also in Figure 4 - Unit test to detect if a counter increment generates a network packet, the test result is shown, proving that the code passed the test.

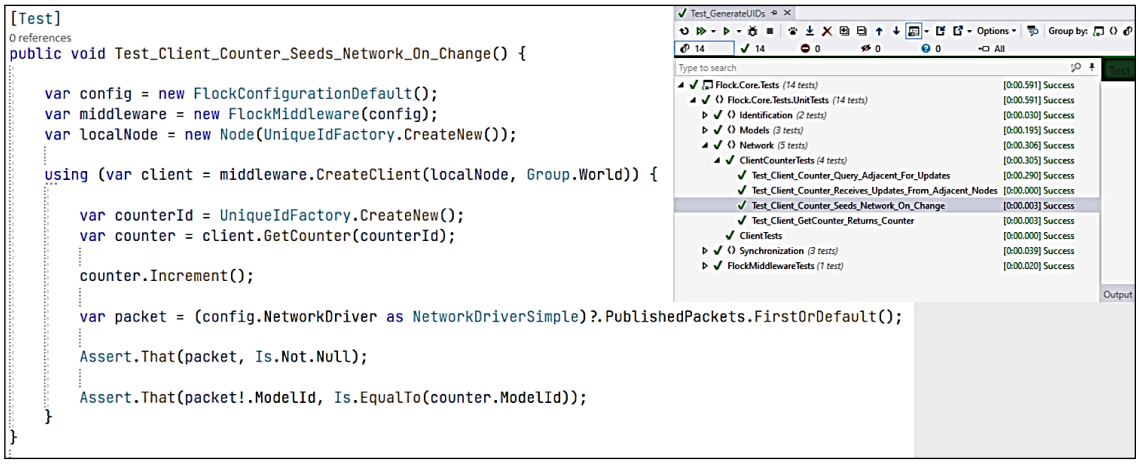

Figure 4 - Unit test to detect if a counter increment generates a network packet

This unit test confirms that an increment in the “Counter” data model, obtained from the “Client”, will cause a network packet to be sent through a “Network Driver” implementation.

6.1. Optimization

Due to the time constraints of this work, the system still lacks optimization on the synchronization of the data models’ state. Data models that contain a lot of information in their state should be fragmented, and only the missing data should be transmitted, to minimize network traffic and reduce latency. Such optimization can be achieved by using a Merkle Tree (Hash Tree) data structure to identify changes.

A Merkle Tree is a hash-based data structure that is a generalization of a hash list. It is a tree structure in which each leaf node is a hash of a block of data, and each non-leaf node is a hash of its children.

Merkle trees are used in distributed systems for efficient data verification. They are efficient because they use hashes instead of the full data. Hashes are ways of encoding data that are much smaller than the actual data itself. This type of structure can provide a structure to index the data inside the state of a data model with an average complexity of $O(\log_k(n))$ where k is the branching factor, and n is the number of child nodes. [Brilliant.org 2021]

This optimization is essential to achieving a very low latency when synchronization takes place. Instead of transmitting the entire state of a CRDT, an algorithm can split the state into parts, and use the hash of these parts to decide which parts are needed to send to the other nodes in the network.

7. Application Scenario

The developed middleware can be applied in different scenarios, wherever eventual consistency is acceptable as on, for example, some IoT networks.

A utilization example would be in the precision agriculture industry. Devices in such industry, usually are spread through a wide geographic region and need to establish a communication channel for data exchange with each other.

In precision agriculture, it is common that devices containing sensors report, frequently, the acquired data so that other devices can use that data as input for their processing.

This type of communication network established between the nodes has the characteristics of a peer-to-peer structure and often comes with bandwidth constraints, due to the wide coverage range. Another characteristic that is observed in such a scenario, is the occurrence of small data packets with frequent transmissions between devices.

Given the characteristics of the described scenario, it would be possible to apply the Flock middleware to mediate the communication and data synchronization. However, it would be up to the application developers to implement the appropriate drivers, such as the “Storage Driver”, “Cryptography Driver” and “Network Driver”, using the respective code interfaces, according to the available hardware.

Although the precision agriculture application scenario was presented, the developed work can be used in many other situations. Flock is a flexible system by design and can be adopted in the most varied scenarios, as long as the scenario is compatible with the middleware characteristics.

It is important to emphasize that the middleware has the objective to decouple the hardware from the application, providing an abstraction layer that sits in-between. In code, the decoupling is achieved through abstract implementations.

8. Conclusion and Future Work

Using the proposed methodology, based on TDD, the middleware implementation was successful. The C# programming language and the .NET Standard 2.0 provided all the features required to build the proposed system while leaving room for future improvements.

The core of the middleware has been implemented, containing a few data models that initially were deemed important, and possible within the time frame allocated for this

work. However, many improvements can be done to achieve better performance and to allow for more flexibility. Introducing new data models, for example, would give the user more options to adapt the middleware to his specific application domain.

By implementing a few optimizations, the middleware can be further improved, into becoming an excellent option for distributed data storage in peer-to-peer networks.

References

Almeida, P. S., Shoker, A., Baquero, C. (2016) “Delta State Replicated Data Types”, Cornell University, New York, United States of America.

Bapat, S., (1994) “Object-Oriented Networks, Models for Architecture, Operations and Management”, Prentice-Hall International, United States of America.

Bartolomeu, C., (2015) “Large-Scale Geo-Replicated Conflict-free Replicated Data Types”, Instituto Superior Técnico, Lisboa, Portugal.

Bender, J., McWherter, J. (2011) “Professional Test-Driven Development with C#: Developing Real World Applications with TDD”, Wiley Publishing, Inc., Indianapolis, Indiana, United States of America.

Brilliant.org, 2021, “Merkle Tree”, Available in: https://brilliant.org/wiki/merkle-tree/, Accessed in November 2021.

Elmasri, R., Navathe, S. B. (2011) “Fundamentals of Database Systems”, 6th edition, Addison-Wesley, Boston, Massachusetts, United States of America.

Etzkorn, L. H., (2017) “Introduction to Middleware: Web Services, Object Components, and Cloud Computing”, CRC Press, Taylor & Francis Group, Boca Raton, Florida, United States of America.

Genc, H., Zu, Y., Chin, T., Halpern, M., Reddi, V. J. (2017) “Flying IoT: Toward Low-Power Vision in the Sky”, “IEEE Micro”, IEEE Computer Society, United States of America, v. 37, n. 6, p. 40-51, nov./dec. 2017.

Hanes, D., Salgueiro, G., Grossetete, P., Barton, R., Henry, J. (2017) “IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things”, 1st edition, Cisco Press, Indianapolis, Indiana, United States of America.

Investopedia (2021) “Blockhain: Merkle Tree”, Available in: https://www.investopedia.com/terms/m/merkle-tree.asp, Accessed in November 2021.

Leitão, J., Pereira, J., Rodrigues, L. (2010) “Gossip-Based Broadcast”, In: Shen, X. S., Yu, H., Buford, J., Akon, M., “Handbook of Peer-to-Peer Networking”, Springer, Boston, Massachusetts, United States of America.

Mascolo, C., Capra, L., Zachariadis, S., Emmerich, W. (2001) “XMIDDLE: A Data-Sharing Middleware for Mobile Computing”, Dept. of Computer Science, University College London, London, United Kingdom.

Mukil, S. (2020) “Anti-Entropy using CRDTs on HA Datastores”, Available in: https://www.infoq.com/presentations/netflix-crdt-entropy/, Accessed in May 2021.

Microsoft Docs (2019) “Assemblies in .NET”, Available in: https://docs.microsoft.com/en-us/dotnet/standard/assembly/, Accessed in April 2021.

Microsoft .NET (2021) “.NET Standard”, https://dotnet.microsoft.com/platform/dotnet-standard, Accessed in April 2021.

Offline First (2021) “Case Studies and Offline First Success Stories”, Available in: http://offlinefirst.org/casestudies/, Accessed in May 2021.

Oracle (2021a) “What Is the Internet of Things (IoT)?”, Available in: https://www.oracle.com/internet-of-things/what-is-iot/, Accessed in April 2021.

Oracle (2021b) “Fusion middleware concepts guide”, Available in: https://docs.oracle.com/cd/E21764_01/core.1111/e10103/intro.htm#ASCON109, Accessed in April 2021.

OrbitDB (2021) “About OrbitDB”, Available in: https://orbitdb.org/about/, Accessed in May 2021.

Pimentel, S. (2015) “The rise of the multimodel database”, Available in: https://www.infoworld.com/article/2861579/the-rise-of-the-multimodel-database.html, Accessed in April 2021.

Riak (2021) “About Riak”, Available in: https://riak.com/about/, Accessed in May 2021.

Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M. (2011) “Conflict-free Replicated Data Types”, 13th International Symposium Stabilization, Safety, and Security of Distributed Systems, Grenoble, France.

Steinmetz, R., Wehrle, K. (2005) “Part I. Peer-to-Peer: Notion, Areas, History and Future”, In: Steinmetz, R., Wehrle, K., “Peer-to-Peer Systems and Applications”, Springer-Verlag Berlin Heidelberg, Germany.

Tanenbaum, A. S., Wetherall, D. J. (2011) “Computer Networks”, 5th edition, Prentice-Hall, United States of America.

Tarkoma, S. (2010) “Overlay Networks: Toward Information Networking”, CRC Press, Boca Raton, Florida, United States of America.

Vogels, W. (2009) “Eventually Consistent”, “Communications of the ACM”, v. 52, n. 1, p. 40-44, Jan. 2009.

Vu, Q. H., Lupu, M., Ooi, B. C. (2010) “Peer-to-Peer Computing: Principles and Applications”, Springer-Verlag Berlin Heidelberg, Germany.