Illustrating Conflict free Replicated Data types (CRDTs)

ITA5006 - Distributed Operating Systems

PROJECT BASED COMPONENT REPORT

by

Kamran Ansari

(22MCA0223)

Bhooshan Birje

(22MCA0233)

School of Information Technology and Engineering

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

JUNE 2023

DECLARATION

I hereby declare that the report entitled “**Illustrating Conflict free Replicated Data types (CRDTs)”** submitted by me, for the ITA5006 Distributed Operating System (EPJ) to Vellore Institute of Technology is a record of bonafide work carried out by me under the supervision of **Dr. T. Senthil Kumar.**

I further declare that the work reported in this report has not been submitted and will not be submitted, either in part or in full, for any other courses in this institute or any other institute or university.

Place : Vellore

Date : 14/06/2023

Signature of the Candidates

Kamran Ansari (22MCA0223)

Bhooshan Birje (22MCA0233)

CONTENTS

Page No.

Acknowledgement 1

Abstract 2

Table of Contents 3

1 **INTRODUCTION** 4

1.1 Objective 6

1.2 Motivation 7

2 **LITERATURE SURVEY** 8

3 **TECHNICAL SPECIFICATIONS** 9

4 **DESIGN** 10

5 **PROPOSED SYSTEM** 12

6 **RESULTS AND DISCUSSION** 16

7 **CONCLUSION** 23

REFERENCES 25

Acknowledgement

I would like to express my heartfelt gratitude and appreciation to Dr. T Senthil Kumar for his invaluable guidance and support throughout the duration of this project. His expertise, dedication, and unwavering commitment to excellence have been instrumental in the successful completion of this endeavor.

I am sincerely grateful to Dr. T Senthil Kumar for providing me with the opportunity to work under his supervision. His vast knowledge and experience in the field have been a constant source of inspiration and motivation. His guidance and mentorship have not only helped me navigate the complexities of the project but have also nurtured my personal and professional growth.

I would like to acknowledge Dr. T Senthil Kumar's patience and willingness to invest time and effort in explaining intricate concepts, reviewing my work, and providing constructive feedback. His insights and suggestions have significantly enriched the quality of the project and have contributed to my development as a researcher.

I extend my heartfelt appreciation to the members of the project team for their collaboration and support. Their contributions and expertise have been invaluable in tackling the challenges encountered during the project. The fruitful discussions and brainstorming sessions with the team have broadened my perspective and enhanced the overall outcomes of this endeavor.

In conclusion, I extend my deepest gratitude to Dr. T Senthil Kumar for his guidance, support, and mentorship throughout this project. His expertise and encouragement have been indispensable, and I am truly fortunate to have had the opportunity to work under his guidance. I am confident that the knowledge and skills acquired through this project will serve as a strong foundation for my future endeavors.

Thank you.

Sincerely,
Kamran Ansari (22MCA0223)
Bhooshan Birje (22MCA0233)

Abstract

Large-scale distributed systems often use data replication across multiple geographic locations to provide increased availability and reduced latency, even in the presence of node and network failures. Some of these systems adopt strong consistency models, where executing an operation requires coordination among a cluster of replicas. While there are ways to improve the throughput of these systems, such as batching, the necessary coordination can lead to high latency for executing a single operation, which depends on the round-trip time between replicas and the chosen protocol. Furthermore, in the event of a network partition or other failures, some nodes may be unable to communicate with the required quorum, resulting in the inability to execute their operations.

Instead of using strong consistency models that require coordination among a quorum of replicas, some distributed systems opt for weaker consistency models such as eventual consistency or causal consistency. Conflict-free Replicated Data Types (CRDT) being an important approach among them. It makes distributed data storage systems and multi-user applications easier to build by simplifying concurrent updates in the data. Conflict-free Replicated Data Types (CRDTs) in distributed systems use optimistic replication approach, where users can change any replica's data without affecting any other source of the replica, even if that replica is offline or disconnected from the others. This approach enables maximum performance and availability.

In this project, we aim to develop and illustrate four of the most prevalent types of CRDTs, including G Counter, PN Counter, G Set, and 2P Set, with the goal of providing an overview of the principles and processes involved in designing CRDTs.

1. INTRODUCTION

A CRDT (Conflict-free Replicated Data Type) is a data structure that is designed to support concurrent modifications and replication in a distributed environment while guaranteeing eventual consistency, even in the presence of network partitions and node failures. CRDTs can be classified into two main types: state-based CRDTs and operation-based CRDTs.

State-based CRDTs

Also known as convergent replicated datatypes, are based on the idea of maintaining a replica of the entire state of the data structure at each node. Each replica can be independently modified, and the modifications can be propagated to other nodes asynchronously. As long as all replicas converge to the same state, the data structure is guaranteed to be eventually consistent.

Operation-based CRDTs

Also known as commutative replicated datatypes, are based on the idea of propagating the operations that modify the data structure rather than the state itself. Each node applies the operations in a consistent order that guarantees convergence, even in the presence of concurrent modifications.

CRDTs can be used to implement a wide range of data structures, such as counters, sets, maps, lists, and more. The specific implementation details depend on the type of CRDT and the requirements of the application.

The main types of CRDT we are going to cover are -

1. **Grow-only Counter**

   A Grow-only Counter is a type of counter that can only increase in value. It is also known as a monotonic counter, meaning that it always moves in one direction (upward) and never decreases.

   Grow-only Counters are often used in distributed systems to track the progress of a system or to assign unique identifiers to events or messages. They can also be used to ensure that certain operations occur only once, by assigning a unique identifier to each operation and using the grow only counter to ensure that each identifier is used only once.

   One important feature of a grow only counter is that it must be designed to be resilient to failures. In a distributed system, multiple processes or nodes may be incrementing the counter simultaneously, so it is important to ensure that the counter remains consistent and does not suffer from race conditions or other types of inconsistencies.

2. **Positive-Negative Counter**

A PN counter is a type of grow-only counter that can be used in distributed systems to track the occurrence of events or updates. It is a type of conflict-free replicated data type (CRDT) that is designed to be resilient to network partitions and other types of failures that may occur in distributed systems.

The name "positive-negative", refers to the fact that a PN counter maintains two values: a positive value that represents the number of "adds" or increments that have been made to the counter, and a negative value that represents the number of "removes" or decrements that have been made to the counter. The current value of the counter is calculated by subtracting the negative value from the positive value.

One key advantage of a PN counter is that it can be updated by any process or node inthe distributed system without requiring coordination or consensus with other nodes. Each node can independently increment or decrement the counter based on its own localevents or updates, and the counter will eventually converge to the same value across allnodes.

3. **Grow-only Set**

The name "G set" stands for "grow-only set", which refers to the fact that a G set can only be modified by adding elements to it. Once an element is added to a G set, it cannot be removed**.**

One key advantage of a G set is that it can be updated by any process or node in the distributed system without requiring coordination or consensus with other nodes. Eachnode can independently add elements to the set based on its own local events or updates,and the set will eventually converge to the same state across all nodes.

To ensure that the G set remains consistent in the face of network partitions and other types of failures, updates are performed using a technique called "convergent replication". This involves merging the local updates from each node in a way that guarantees eventual consistency, even if updates occur concurrently or out of order.

4. **Two-Phase Set**

The 2P set is composed of two separate sets: an "add set" that contains elements that have been added to the set, and a "remove set" that contains elements that have been removed from the set. Elements are added to the add set, and removed from the remove set. When an element is removed, it is not actually deleted from the set; instead, it is moved from the add set to the remove set. This approach ensures that elements are never truly lost, and can be added back to the set at any time.

Each node in the distributed system maintains a copy of both the add set and the removeset. When a node wants to add an element to the set, it adds the element to its local addset and broadcasts this update to all other nodes in the system. When a node wants to remove an element from the set, it adds the element to its local remove set and broadcasts this update to all other nodes in the system.

One limitation of the 2P set is that if an element is added to the set after it has been removed, the element will still be considered removed because it exists in the remove set. This can be addressed by using a more complex data structure that tracks the version history of each element in the set.

1.1 OBJECTIVE

The objective of the project "Illustrating Conflict-Free Replicating Datatypes" is to develop a graphical representation framework for visualizing the behavior of distributed systems and demonstrating the evolution of values within these systems over time. By utilizing this graphical representation, the project aims to showcase the conflict-free nature of replicating datatypes in distributed systems and highlight their ability to handle concurrent updates without compromising data consistency.

The specific objectives of the project include:

- **Designing and implementing a graphical representation framework:** Develop a visual representation framework that effectively portrays the peers in a distributed system and their interactions. This framework should provide a clear visualization of the system's state and how it evolves with the changes in the participating peers.

- **Integrating conflict-free replicating datatypes (CRDTs):** Incorporate CRDTs into the graphical representation framework to demonstrate their conflict-free nature. CRDTs are data structures designed for distributed systems, ensuring that concurrent updates from multiple peers can be reconciled without conflicts.

- **Simulating distributed systems with dynamic updates:** Create scenarios where the values stored in the distributed system are updated dynamically over time. Simulate concurrent updates from multiple peers to showcase the ability of CRDTs to handle conflicts and maintain eventual consistency.

- **Visualizing value changes and conflict resolution:** Display the changes in values within the distributed system using graphical representations. Highlight how conflicts are resolved using the properties of CRDTs, emphasizing the absence of conflicts and the preservation of data consistency.

- **Providing educational materials and documentation:** Produce educational materials, such as tutorials or documentation, that explain the concepts of conflict-free replicating datatypes and their graphical representation. These resources will enable users to understand the benefits and applications of CRDTs in distributed systems.

By accomplishing these objectives, the project aims to enhance the understanding of CRDTs and their practical implementation in distributed systems. The graphical representation framework will serve as a visual aid to effectively illustrate the benefits of conflict-free replicating datatypes and their ability to maintain consistency in dynamic and decentralized environments.

1.2 MOTIVATION

A Conflict-free Replicated Data Type (CRDT) helps to simplify distributed data storage systems and multi-user applications. These systems often require multiple copies of the same data to be stored on different computers, or replicas. Some examples of such systems include mobile apps, distributed databases, collaboration software, and large-scale data storage and processing systems. In these systems, the data may be modified concurrently on different replicas. CRDTs help to manage this by ensuring that updates to the data can be propagated to all replicas without conflicts.

Many systems require multiple computers to store redundant copies of certain of data (known as replicas). These are some instances of such systems:

- mobile applications that need to sync data from the local device (such as calendars, contacts, notes, or reminders) to other devices owned by the same user.
- distributed databases, which have several copies of the data on hand (either in the same data centre or in separate places) to ensure that the system functions properly even if some of the copies are unavailable.
- program for collaboration, such as Google Docs, Trello, Figma, and many others, that allows many users to simultaneously edit the same document or set of data.
- technologies for processing and storing large amounts of data, which replicate data to reach global scalability.

2. LITERATURE SURVEY

Name of the paper	Findings
"A Comprehensive Study of Convergent and Commutative Replicated Data Types" by Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski	• This paper provides an overview of CRDTs and compares different types of CRDTs based on their properties and use cases. It also discusses practical considerations for implementing CRDTs in distributed systems. • Provides an overview of counter and set CRDTs, as well as other types of CRDTs. It compares different counter and set CRDT algorithms and discusses their properties and use cases.
"Efficient Synchronization of State-based CRDTs" by Maysam Yabandeh, Annette Bieniusa, and Carlos Baquero	• This paper presents an algorithm for efficient synchronization of state-based CRDTs, which reduces the amount of network traffic required for synchronization while maintaining strong eventual consistency.
"Mergeable Replicated Data Types" by Sebastian Burckhardt, Manuel Fähndrich, and Daan Leijen	• This paper introduces the concept of mergeable replicated data types (MRDTs), which are a generalization of CRDTs that support more flexible merge operations. The paper provides a framework for designing and implementing MRDTs and discusses several examples.
"Conflict-free Replicated Data Types: An Overview" by Almeida et al	• This paper provides an overview of CRDTs and their various types, along with their key properties and applications. It also provides a survey of different algorithms and techniques for implementing CRDTs.
"A Highly Available Set CRDT using Commutative Replicated Datatypes" by Nuno Preguiça, Marc Shapiro, and Carlos Baquero.	• This paper presents a set CRDT algorithm that guarantees strong eventual consistency and high availability, even in the presence of network partitions and node failures. The algorithm is based on commutative replicated datatypes and uses a timestamp-based reconciliation technique to merge concurrent updates.
"LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing" by Pierre Genevès, Pascal Molli, and Achour Mostefaoui	• This paper presents an algorithm for constructing an ordered sequence CRDT that scales well for large documents and supports efficient operations for insertion, deletion, and navigation. The algorithm is based on a tree structure and uses an adaptive allocation scheme to minimize the number of nodes that need to be transmitted during updates.

3. TECHNICAL SPECIFICATIONS

As this is a web-application, it does not have any special requirements as such. Any computer with the capability to run a browser such as Chrome, Edge or Firefox will be able to experience this project as intended. The proposed web application is not graphics heavy so even low powered computers, and even smartphones can run it.

Hardware Requirements –

- **Processor:** At least a dual-core processor, but a quad-core or higher is recommended for better performance
- **Memory (RAM):** At least 2 GB of RAM, but 4 GB or higher is recommended for better performance.

Software requirements –

- **Operating system**: An operating system that can run a compatible web browser
- **Web browser:** The web application should be compatible with the latest versions of popular web browsers such as Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge.
- **Programming languages:** The web application requires specific programming languages such as HTML, CSS, JavaScript, Typescript and ReactJS.
- **Technologies used:** Next.js, ReactJS, NodeJS
- **Dependencies:** Following dependencies installed from npm –

```
"dependencies": {
      "@emotion/react": "^11.11.0",
      "@emotion/styled": "^11.11.0",
      "@mui/material": "^5.13.1",
      "next": "13.3.1",
      "react": "18.2.0",
       "react-dom": "18.2.0"
},
"devDependencies": {
      "@swc/core": "^1.3.58",
      "@swc/jest": "^0.2.26",
      "@types/jest": "^29.5.1",
      "@types/node": "20.1.5",
      "@types/react": "18.2.6",
      "jest": "^29.5.0",
      "typescript": "5.0.4"
  }
```

4. DESIGN

A ReactJS application runs on the user's browser and provides the interface the interactive illustration of CRDTs. Which themselves are implemented in Typescript, along with their respective testing code.

There are four different pages for each CRDT but the overall functionality is similar in all of them. Each of them have a collection of system which in which –

a. Systems can be added.

b. Value of systems can be changed independently.

c. Systems can be merged.

4.1 BLOCK DIAGRAM

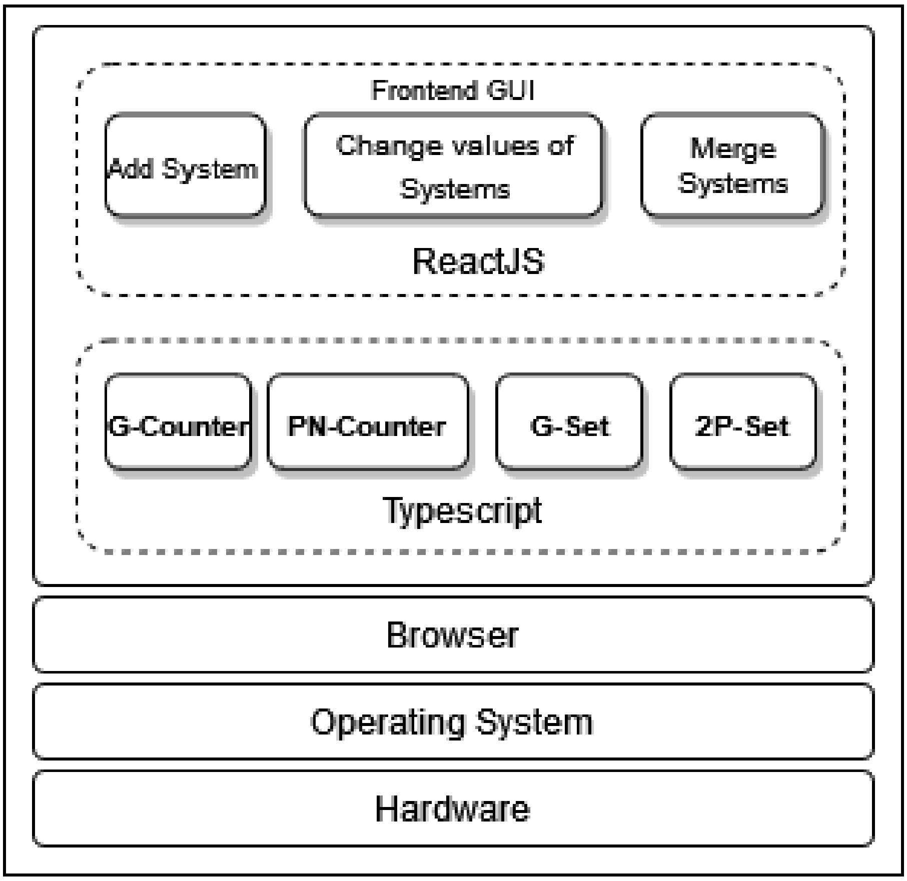


Figure 1 Block Diagram of Application

4.2 FLOWCHART

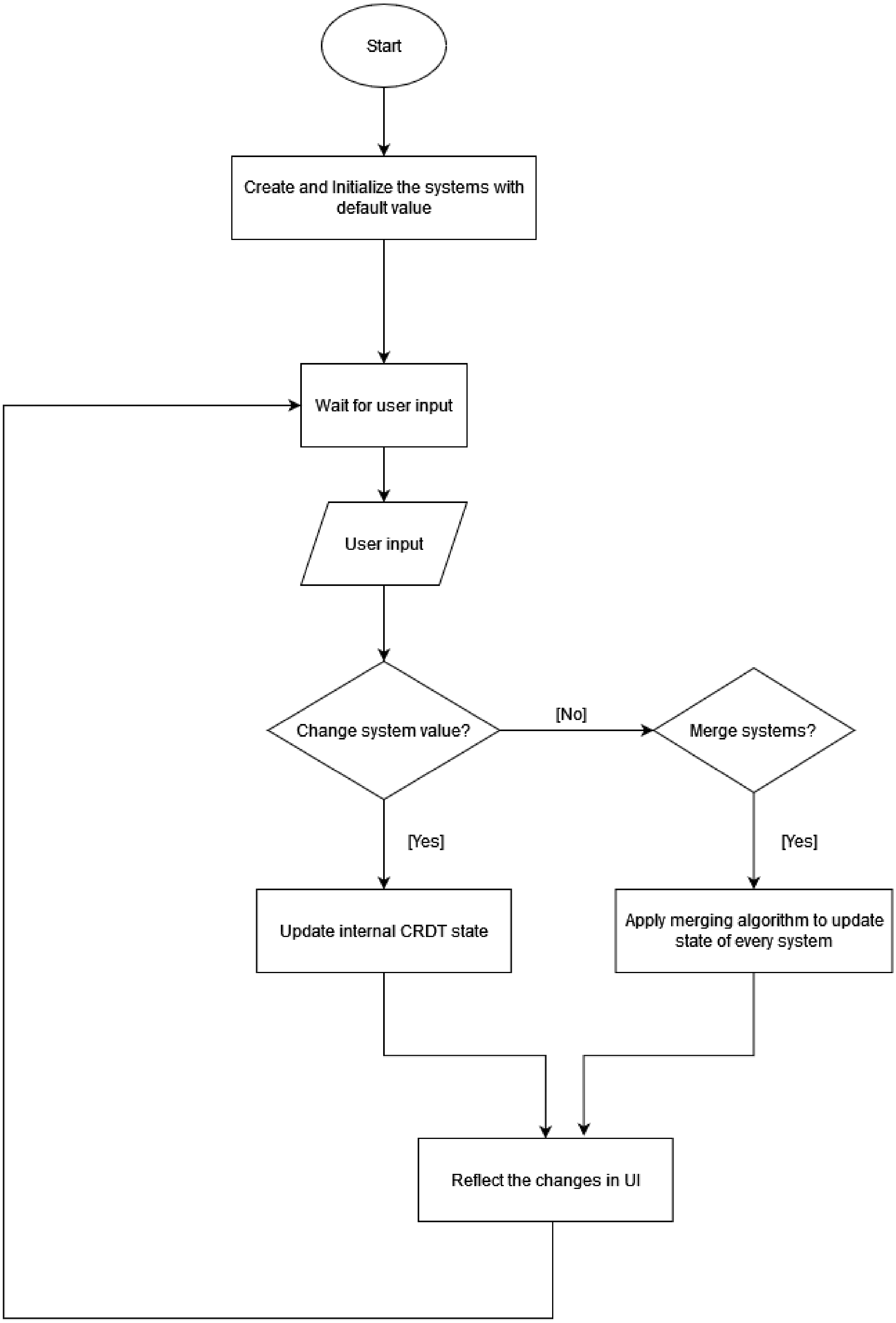

Start
Create and Initialize the systems with default value
Wait for user input
User input
Change system value?
[No]
Merge systems?
[Yes]
[Yes]
Update internal CRDT state
Apply merging algorithm to update state of every system
Reflect the changes in UI

Figure 2 Flowchart of Application

5. PROPOSED SYSTEM

To illustrate the concepts of Conflict-free Replicated Data Types (CRDTs), we have used web technologies. The project follows a comprehensive methodology that combines theoretical understanding, practical implementation, and effective visualization techniques. The proposed methodology comprises the following key steps:

1. **Research and Understanding:**

   a. Conduct an in-depth literature review to acquire a solid theoretical foundation of CRDTs, including their concepts, principles, and different types.

   b. Explore existing web technologies commonly used for building interactive and real-time applications, such as ReactJS and Next.js.

2. **Design and Planning:**

   a. Define the scope and objectives of the project, including the specific CRDT types to be illustrated and the target audience.

   b. Identify the essential features and functionalities to be implemented in the web-based CRDT visualization tool.

   c. Create a detailed design plan, including the user interface (UI) layout, data structures, and interactions.

   d. Design the proposed user interface in prototyping tools such as Figma.

3. **Implementing and Testing CRDTs:**

   a. Implement chosen CRDTS in Typescript using programming concepts such as classes and functions.

   b. Write tests for those CRDTs to ensure error free working of the datatypes.

4. **Development of the Web Application:**

   a. Utilize web development frameworks, libraries, and tools to build the interactive web application.

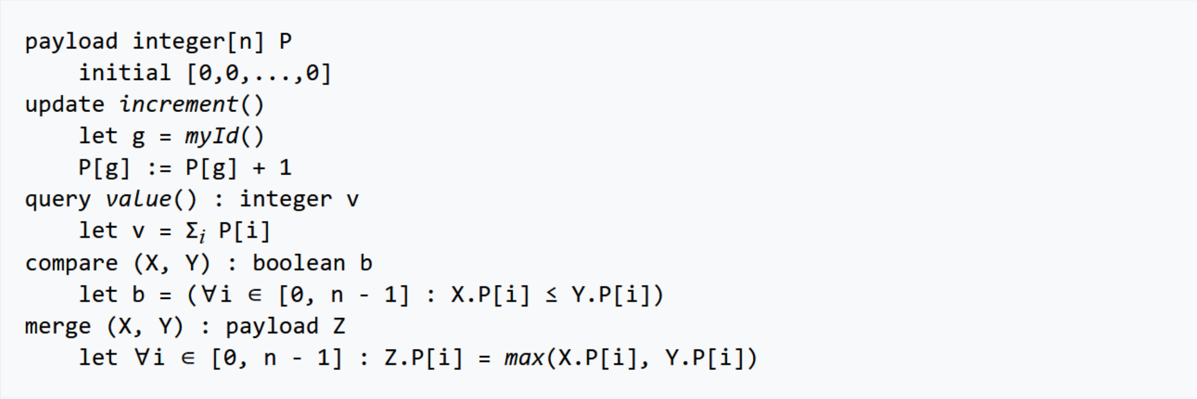
b. Implement the necessary ReactJS components to create an intuitive and visually appealing user interface.

c. Incorporate CRDT algorithms and logic into the web application, ensuring proper data synchronization and conflict resolution mechanisms.

5.1 PROPOSED ALGORITHMS

GCounter

- The *internal state* of a G-Counter replicated on n machines is n-length array of non-negative integers.
- The `query` method returns the sum of every element in the n-length array.
- The `add(x)` update method, when invoked on the ith server, increments the ith entry of the n-length array by `x`. For example, server 0 will increment the 0th entry of the array, server 1 will increment the 1st entry of the array, and so on.
- The `merge` method performs a pairwise maximum of the two arrays.



```
payload integer[n] P
    initial [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
query value() : integer v
    let v = $\Sigma_i$ P[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

PNCounter

- The *internal state* of a PN-Counter is a pair of two G-Counters named `p` and `n`. `p` represents the total value added to the PN-Counter while `n` represents the total value subtracted from the PN-Counter.
- The `query` method returns the difference `p.query() - n.query()`.
- The `add(x)` method (the first of the two update methods) invokes `p.add(x)`.
- The `sub(x)` method (the second of the two update methods) invokes `n.add(x)`.
- The `merge` method performs a pairwise merge of `p` and `n`.

```
payload integer[n] P, integer[n] N
    initial [0,0,...,0], [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
update decrement()
    let g = myId()
    N[g] := N[g] + 1
query value() : integer v
    let v = $\Sigma_i$ P[i] - $\Sigma_i$ N[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i] ∧ ∀i ∈ [0, n - 1] : X.N[i] ≤ Y.N[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
    let ∀i ∈ [0, n - 1] : Z.N[i] = max(X.N[i], Y.N[i])
```

GSet

- The *internal state* of a G-Set is just a set.
- The `query` method returns the set.
- The `add(x)` update method adds `x` to the set.
- The `merge` method performs a set union.

```
payload set A
    initial ∅
update add(element e)
    A := A ∪ {e}
query lookup(element e) : boolean b
    let b = (e ∈ A)
compare (S, T) : boolean b
    let b = (S.A ⊆ T.A)
merge (S, T) : payload U
    let U.A = S.A ∪ T.A
```

2P-Set (Two-Phase Set)

- The *internal state* of a 2P-Set is a pair of two G-Sets named `a` and `r`. `a` represents the set of values added to the 2P-Set while `r` represents the set of values removed from the 2P-Set.
- The `query` method returns the set difference `a.query() - r.query()`.
- The `add(x)` method (the first of the two update methods) invokes `a.add(x)`.
- The `sub(x)` method (the second of the two update methods) invokes `r.add(x)`.
- The `merge` method performs a pairwise merge of `a` and `r`.

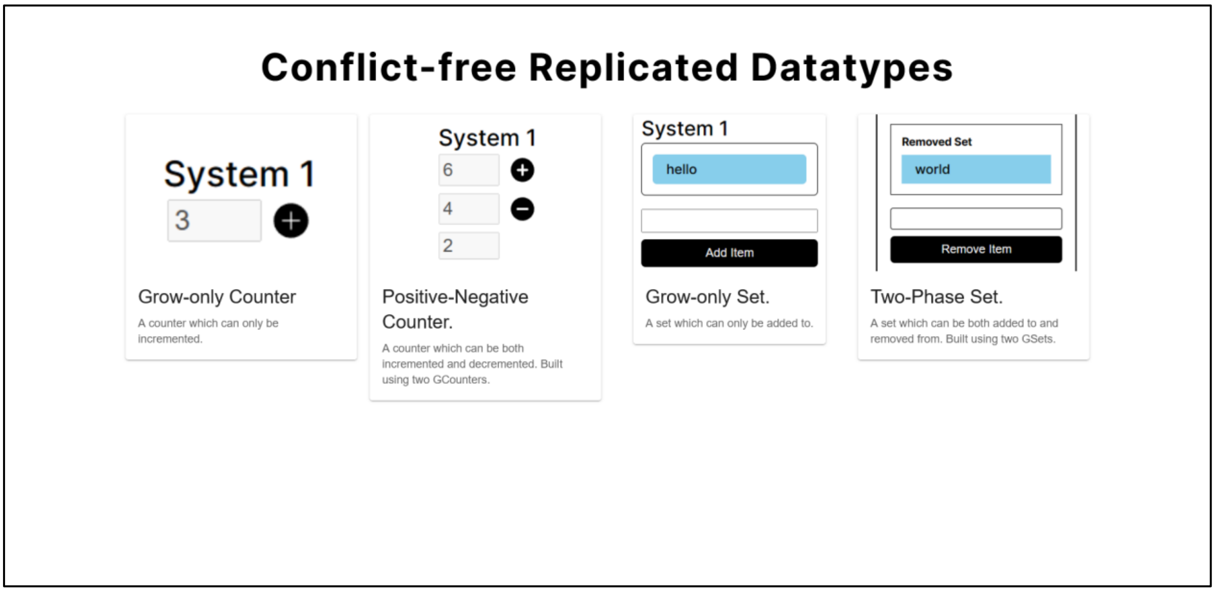
```
payload set A, set R
    initial ∅, ∅
query lookup(element e) : boolean b
    let b = (e ∈ A ∧ e ∉ R)
update add(element e)
    A := A ∪ {e}
update remove(element e)
    pre lookup(e)
    R := R ∪ {e}
compare (S, T) : boolean b
    let b = (S.A ⊆ T.A ∧ S.R ⊆ T.R)
merge (S, T) : payload U
    let U.A = S.A ∪ T.A
    let U.R = S.R ∪ T.R
```

6. RESULT AND DISCUSSION

The project successfully achieved its objectives by effectively illustrating conflict-free replicating datatypes using graphical representations of peers in a distributed system. The key findings demonstrated the feasibility and benefits of this approach. The visualizations provided a clear and intuitive understanding of how values change over time in a distributed system.

The project showcased the potential of CRDTs in enabling replication and consistency without the need for centralized coordination. The ability to visualize real-time value changes and observe the convergence of the system greatly enhanced the understanding of distributed systems and CRDTs.

6.1 OUTPUT SCREENSHOTS



Conflict-free Replicated Datatypes

Grow-only Counter

A counter which can only be incremented.

Positive-Negative Counter.

A counter which can be both incremented and decremented. Built using two GCounters.

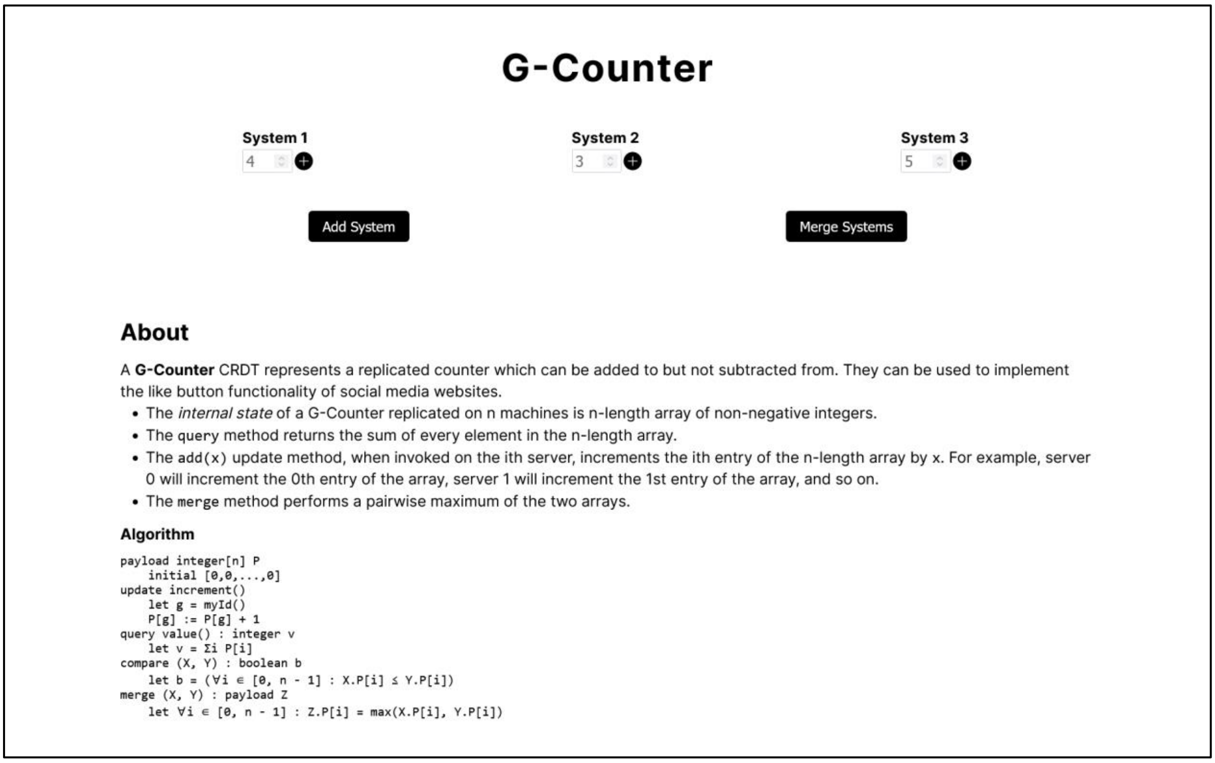
Grow-only Set.

A set which can only be added to.

Two-Phase Set.

A set which can be both added to and removed from. Built using two GSets.

Figure 3 Main page of Web Application



G-Counter

System 1

4

System 2

3

System 3

5

About

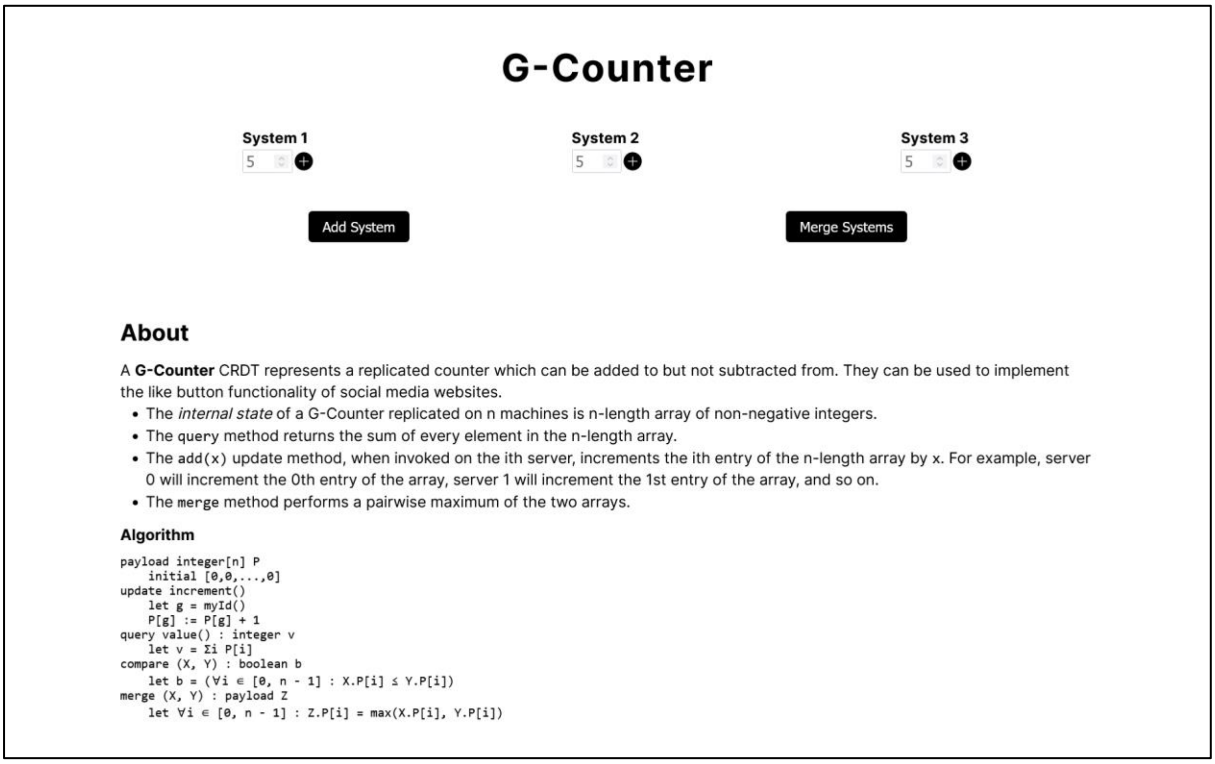
A **G-Counter** CRDT represents a replicated counter which can be added to but not subtracted from. They can be used to implement the like button functionality of social media websites.

- The *internal state* of a G-Counter replicated on n machines is n-length array of non-negative integers.
- The `query` method returns the sum of every element in the n-length array.
- The `add(x)` update method, when invoked on the ith server, increments the ith entry of the n-length array by x. For example, server 0 will increment the 0th entry of the array, server 1 will increment the 1st entry of the array, and so on.
- The `merge` method performs a pairwise maximum of the two arrays.

Algorithm

```
payload integer[n] P
    initial [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
query value() : integer v
    let v = Σi P[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

Figure 4 G Counter - Independent States



G-Counter

System 1

5

System 2

5

System 3

5

About

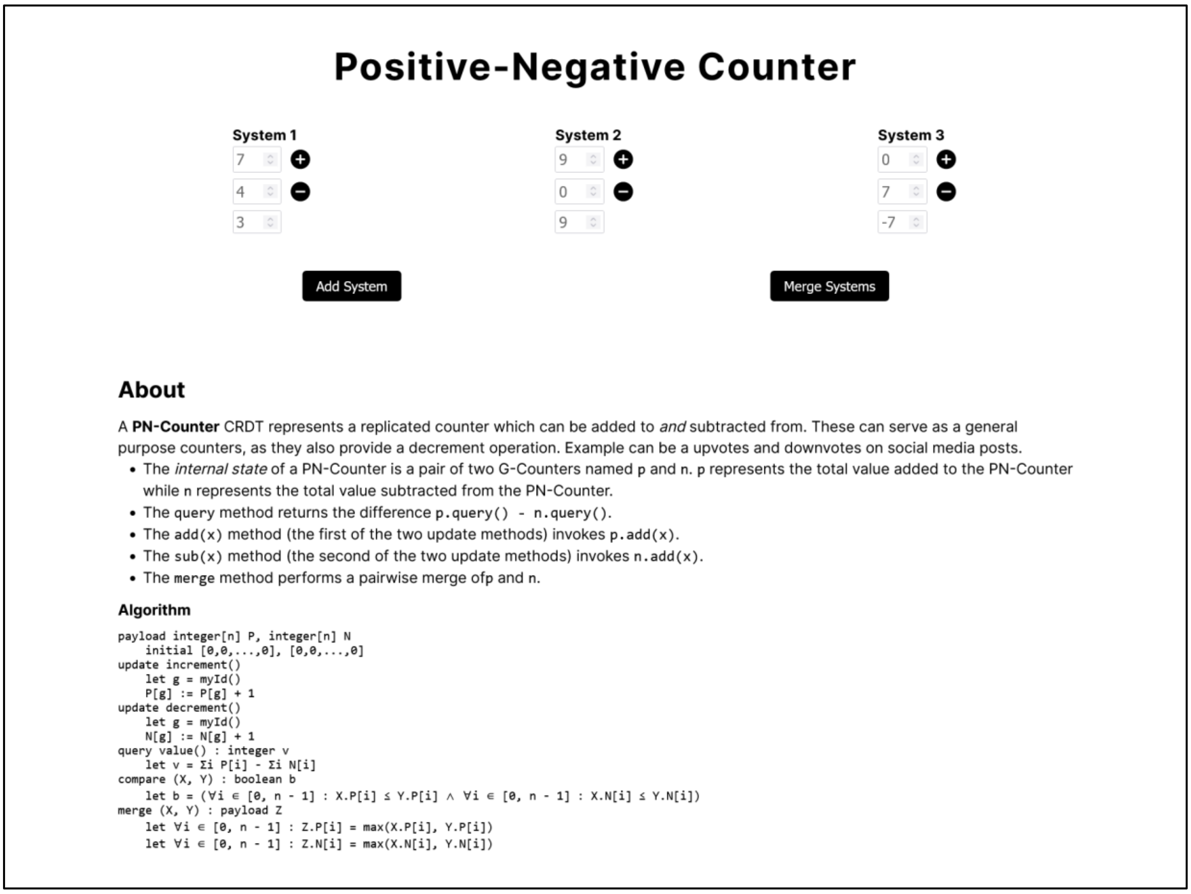
A **G-Counter** CRDT represents a replicated counter which can be added to but not subtracted from. They can be used to implement the like button functionality of social media websites.

- The *internal state* of a G-Counter replicated on n machines is n-length array of non-negative integers.
- The `query` method returns the sum of every element in the n-length array.
- The `add(x)` update method, when invoked on the ith server, increments the ith entry of the n-length array by x. For example, server 0 will increment the 0th entry of the array, server 1 will increment the 1st entry of the array, and so on.
- The `merge` method performs a pairwise maximum of the two arrays.

Algorithm

```
payload integer[n] P
    initial [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
query value() : integer v
    let v = Σi P[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

Figure 5 G-Counter- Merged States



Positive-Negative Counter

System 1
7
4
3

System 2
9
0
9

System 3
0
7
-7

About

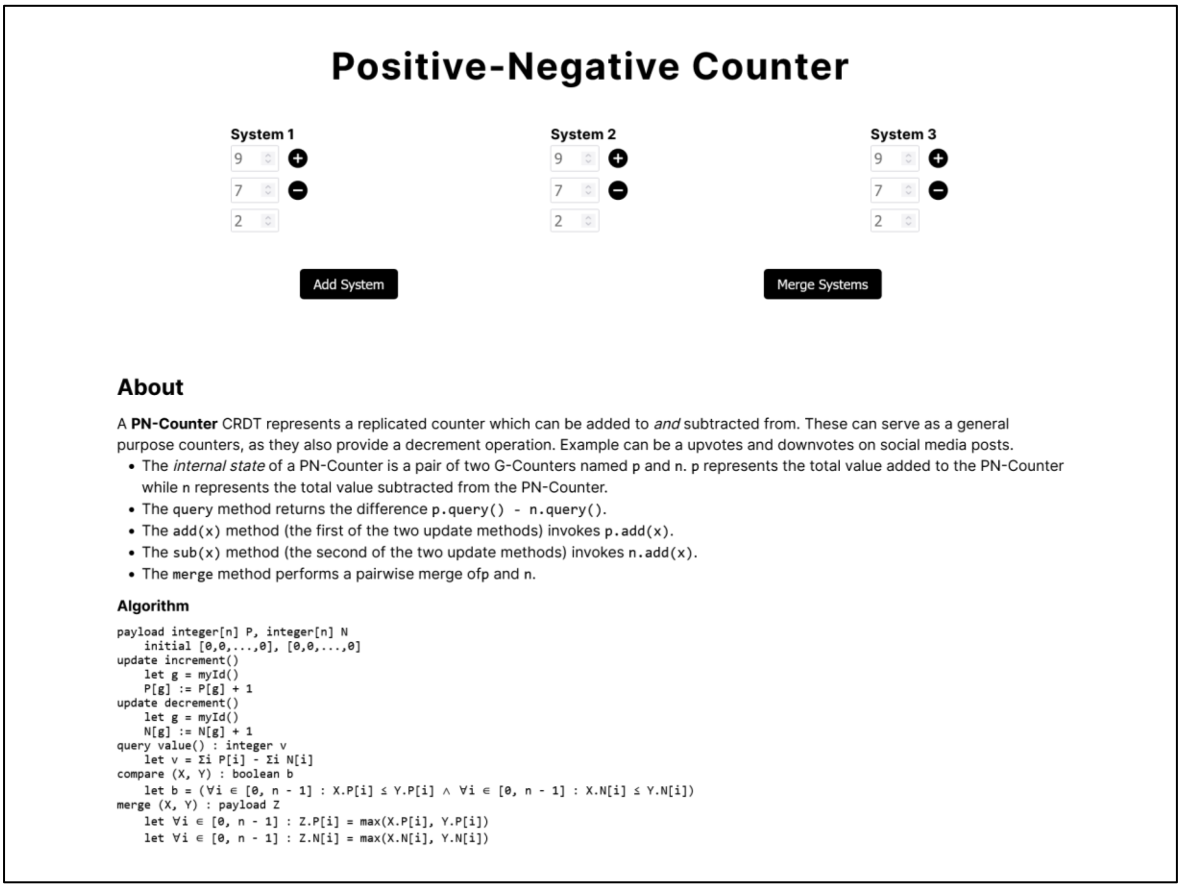
A **PN-Counter** CRDT represents a replicated counter which can be added to *and* subtracted from. These can serve as a general purpose counters, as they also provide a decrement operation. Example can be a upvotes and downvotes on social media posts.

- The *internal state* of a PN-Counter is a pair of two G-Counters named p and n. p represents the total value added to the PN-Counter while n represents the total value subtracted from the PN-Counter.
- The query method returns the difference p.query() - n.query().
- The add(x) method (the first of the two update methods) invokes p.add(x).
- The sub(x) method (the second of the two update methods) invokes n.add(x).
- The merge method performs a pairwise merge ofp and n.

Algorithm

```
payload integer[n] P, integer[n] N
    initial [0,0,...,0], [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
update decrement()
    let g = myId()
    N[g] := N[g] + 1
query value() : integer v
    let v = Σi P[i] - Σi N[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i] ∧ ∀i ∈ [0, n - 1] : X.N[i] ≤ Y.N[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
    let ∀i ∈ [0, n - 1] : Z.N[i] = max(X.N[i], Y.N[i])
```

Figure 6 PN-Counter - Independent State



Positive-Negative Counter

System 1
9
7
2

System 2
9
7
2

System 3
9
7
2

About

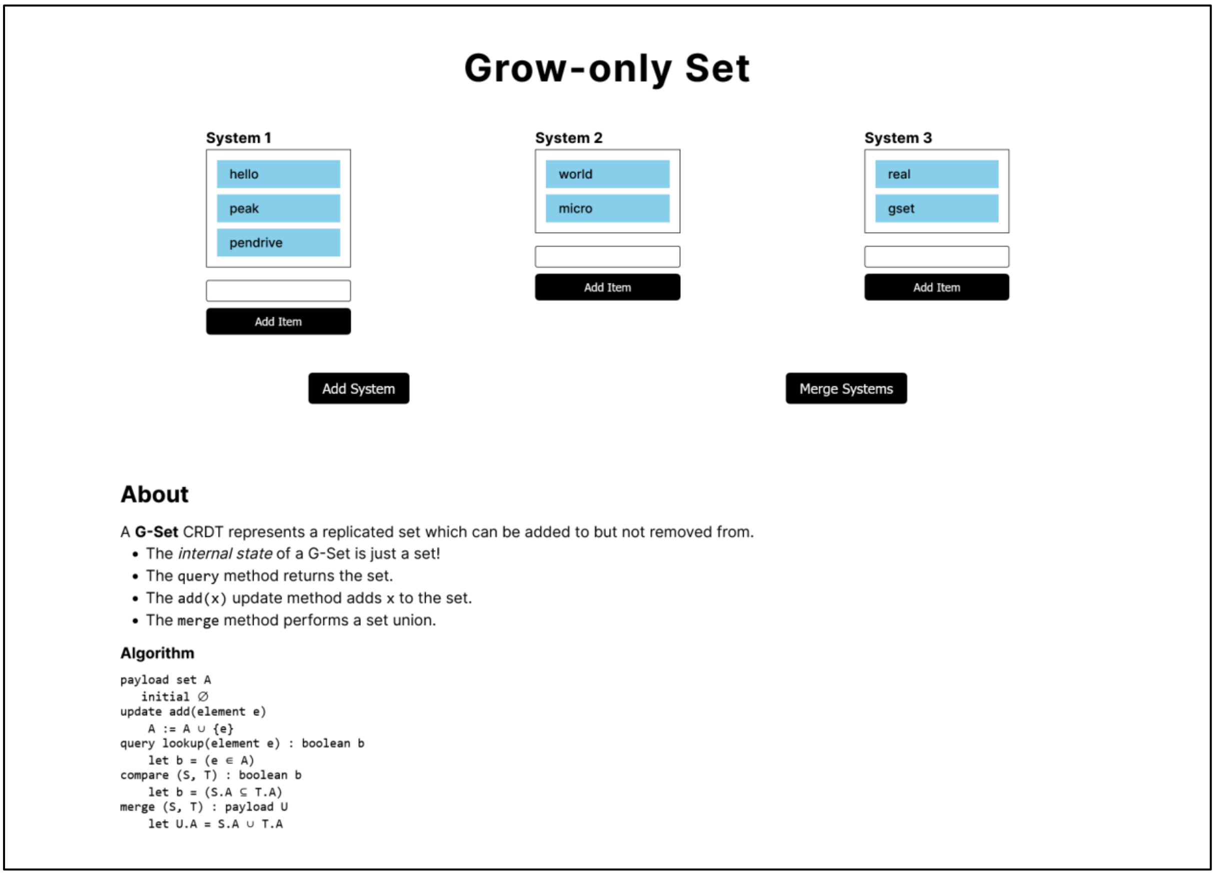
A **PN-Counter** CRDT represents a replicated counter which can be added to *and* subtracted from. These can serve as a general purpose counters, as they also provide a decrement operation. Example can be a upvotes and downvotes on social media posts.

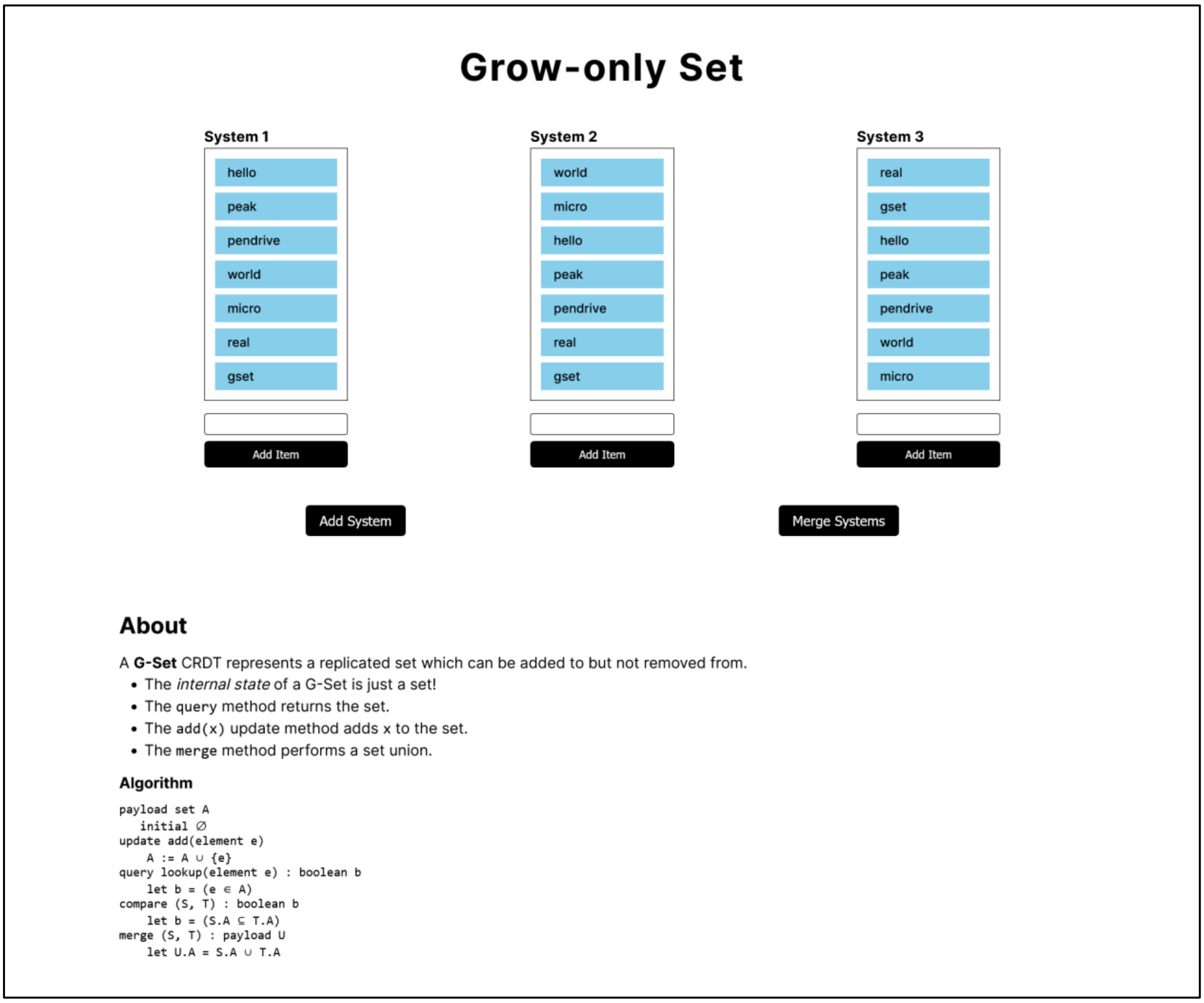
- The *internal state* of a PN-Counter is a pair of two G-Counters named p and n. p represents the total value added to the PN-Counter while n represents the total value subtracted from the PN-Counter.
- The query method returns the difference p.query() - n.query().
- The add(x) method (the first of the two update methods) invokes p.add(x).
- The sub(x) method (the second of the two update methods) invokes n.add(x).
- The merge method performs a pairwise merge ofp and n.

Algorithm

```
payload integer[n] P, integer[n] N
    initial [0,0,...,0], [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
update decrement()
    let g = myId()
    N[g] := N[g] + 1
query value() : integer v
    let v = Σi P[i] - Σi N[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i] ∧ ∀i ∈ [0, n - 1] : X.N[i] ≤ Y.N[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
    let ∀i ∈ [0, n - 1] : Z.N[i] = max(X.N[i], Y.N[i])
```

Figure 7 PN-Counter - Merged States





Grow-only Set

About

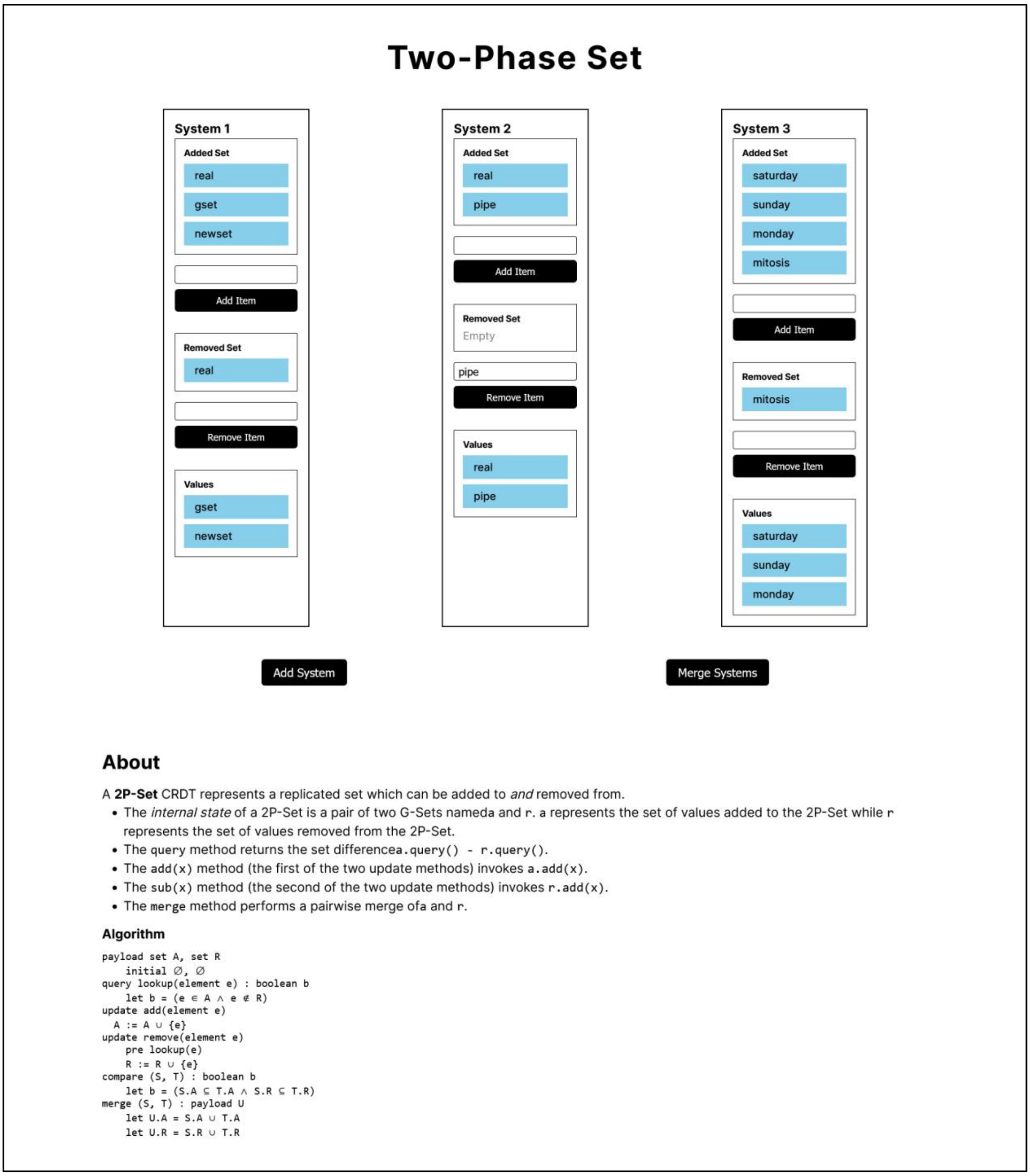
A **G-Set** CRDT represents a replicated set which can be added to but not removed from.

- The *internal state* of a G-Set is just a set!
- The `query` method returns the set.
- The `add(x)` update method adds x to the set.
- The `merge` method performs a set union.

Algorithm

```
payload set A
   initial ∅
update add(element e)
    A := A ∪ {e}
query lookup(element e) : boolean b
    let b = (e ∈ A)
compare (S, T) : boolean b
    let b = (S.A ⊆ T.A)
merge (S, T) : payload U
    let U.A = S.A ∪ T.A
```

Figure 9 GSet - Merged States



About

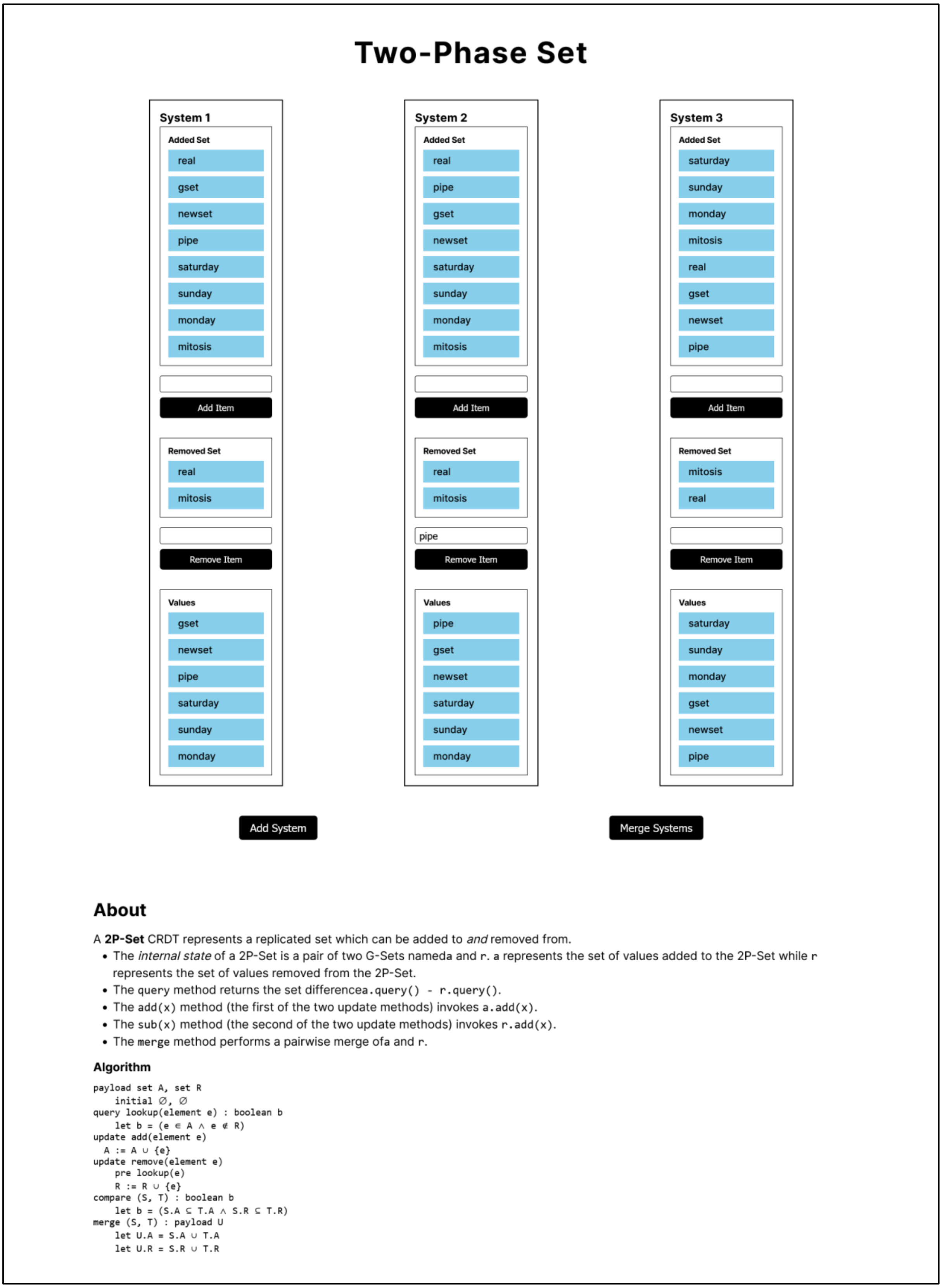
A **2P-Set** CRDT represents a replicated set which can be added to *and* removed from.

- The *internal state* of a 2P-Set is a pair of two G-Sets nameda and r. a represents the set of values added to the 2P-Set while r represents the set of values removed from the 2P-Set.
- The query method returns the set differencea.query() - r.query().
- The add(x) method (the first of the two update methods) invokes a.add(x).
- The sub(x) method (the second of the two update methods) invokes r.add(x).
- The merge method performs a pairwise merge ofa and r.

Algorithm

```
payload set A, set R
    initial ∅, ∅
query lookup(element e) : boolean b
    let b = (e ∈ A ∧ e ∉ R)
update add(element e)
  A := A ∪ {e}
update remove(element e)
    pre lookup(e)
    R := R ∪ {e}
compare (S, T) : boolean b
    let b = (S.A ⊆ T.A ∧ S.R ⊆ T.R)
merge (S, T) : payload U
    let U.A = S.A ∪ T.A
    let U.R = S.R ∪ T.R
```

Figure 10 2PSet - Independent States



Two-Phase Set

System 1

Added Set
- real
- gset
- newset
- pipe
- saturday
- sunday
- monday
- mitosis

Removed Set
- real
- mitosis

Values
- gset
- newset
- pipe
- saturday
- sunday
- monday

System 2

Added Set
- real
- pipe
- gset
- newset
- saturday
- sunday
- monday
- mitosis

Removed Set
- real
- mitosis

pipe

Values
- pipe
- gset
- newset
- saturday
- sunday
- monday

System 3

Added Set
- saturday
- sunday
- monday
- mitosis
- real
- gset
- newset
- pipe

Removed Set
- mitosis
- real

Values
- saturday
- sunday
- monday
- gset
- newset
- pipe

About

A **2P-Set** CRDT represents a replicated set which can be added to *and* removed from.

- The *internal state* of a 2P-Set is a pair of two G-Sets nameda and `r`. `a` represents the set of values added to the 2P-Set while `r` represents the set of values removed from the 2P-Set.
- The `query` method returns the set differencea`.query() - r.query()`.
- The `add(x)` method (the first of the two update methods) invokes `a.add(x)`.
- The `sub(x)` method (the second of the two update methods) invokes `r.add(x)`.
- The `merge` method performs a pairwise merge ofa and `r`.

Algorithm

```
payload set A, set R
    initial ∅, ∅
query lookup(element e) : boolean b
    let b = (e ∈ A ∧ e ∉ R)
update add(element e)
  A := A ∪ {e}
update remove(element e)
    pre lookup(e)
    R := R ∪ {e}
compare (S, T) : boolean b
    let b = (S.A ⊆ T.A ∧ S.R ⊆ T.R)
merge (S, T) : payload U
    let U.A = S.A ∪ T.A
    let U.R = S.R ∪ T.R
```

Figure 11 2PSet - Merged States

7. CONCLUSION

In conclusion, we are thrilled to announce the successful completion of our project, which aimed to create a graphical representation of the working principles of Conflict-free Replicated Data Types (CRDTs) - specifically the G-counter, PN-counter, G-set, and 2p-set. Leveraging the power of ReactJS, TypeScript libraries, and Jest for testing, we have developed an interactive and visually captivating user interface that effectively demonstrates the functionality and resilience of these CRDTs.

Throughout the project, we embarked on a comprehensive exploration of CRDTs, delving into the theoretical foundations and practical applications of G-counter, PN-counter, G-set, and 2p-set. By thoroughly understanding the algorithms behind these CRDTs, we were able to translate their complex mechanisms into a visually engaging format.

Using ReactJS, we meticulously crafted an intuitive user interface that allows users to interact with the implemented CRDTs. The graphical representation provides a clear visualization of data replication processes, concurrent updates, and eventual consistency. Users can witness the dynamic behaviour of the CRDTs in real-time, gaining a deeper understanding of their operation.

We harnessed the capabilities of TypeScript and its libraries to develop a robust and type-safe implementation of the CRDTs. This ensured the reliability and correctness of our application, allowing for smooth interactions and accurate results. By employing Jest for testing, we were able to create an extensive test suite, covering various scenarios and edge cases. Rigorous testing enabled us to identify and resolve any potential issues, ensuring the accuracy and stability of our implementation.

The successful completion of this project has not only enhanced our understanding of CRDTs but also honed our skills in developing complex graphical interfaces using ReactJS and TypeScript. We have gained invaluable experience in translating abstract concepts into tangible visualizations, enabling users to grasp the intricate workings of distributed data replication.

In conclusion, our project has achieved its objectives by providing an immersive and visually appealing experience that demonstrates the functionality of CRDTs - G-counter, PN-counter, G-set, and 2p-set. The combination of ReactJS, TypeScript libraries, and Jest testing has empowered us to create an application that seamlessly showcases the power and resilience of these CRDTs. We believe that our project will serve as a valuable educational resource, fostering a deeper understanding of CRDTs and their role in distributed systems.

We are proud of the effort and dedication invested by our team in successfully completing this project. By contributing to the advancement of CRDT research, we hope to inspire further exploration and innovation in the field of distributed systems.

REFERENCES

1. "A Comprehensive Study of Convergent and Commutative Replicated Data Types" by Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski
2. "Efficient Synchronization of State-based CRDTs" by Maysam Yabandeh, Annette Bieniusa, and Carlos Baquero
3. "Conflict-free Replicated Data Types: An Overview" by Almeida et al
4. "Mergeable Replicated Data Types" by Sebastian Burckhardt, Manuel Fähndrich, and Daan Leijen
5. "A Highly Available Set CRDT using Commutative Replicated Datatypes" by Nuno Preguiça, Marc Shapiro, and Carlos Baquero.
6. "LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing" by Pierre Genevès, Pascal Molli, and Achour Mostefaoui
7. Georges Younes, Paulo Sérgio Almeida, and Carlos Baquero. Compact resettable counters through causal stability. In 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2017. ACM, April 2017
8. Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and João Leitão. Borrowing an identity for a distributed counter: Work in progress report. In 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2017. ACM, April 2017.
9. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In 13th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS 2011, pages 386--400. Springer LNCS volume 6976, October 2011
10. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report 7506, INRIA, January 2011.
11. Weihai Yu, Victorien Elvinger, and Claudia-Lavinia Ignat. A generic undo support for state-based CRDTs. In 23rd International Conference on Principles of Distributed Systems, OPODIS 2019. Dagstuhl LIPIcs, November 2019.