IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 15 March 2025, accepted 5 May 2025, date of publication 12 May 2025, date of current version 19 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3568925

RESEARCH ARTICLE

Crust: A Modular Framework for Conflict-Free Replicated Data Types (CRDTs) Development, Validation, and Benchmarking

YUNRUI ZHU AND JING MA, (Member, IEEE)
School of Engineering, Computer and Mathematical Sciences, Auckland University of Technology, Auckland 1010, New Zealand
Corresponding author: Jing Ma (jing.ma@aut.ac.nz)

ABSTRACT Conflict-free Replicated Data Types (CRDTs) are vital for achieving strong eventual consistency in distributed systems, but their development and evaluation face significant challenges by inadequate tooling. While CRDT research focus on algorithms and applications, critical gaps persist in validation and performance benchmarking. To address this, a novel Rust-based framework "Crust" designed to offer a modular, configurable, and extensible platform for developing, validating, and benchmarking CRDT implementations. Crust includes Core, Config, Network, Validation, and Benchmark moduels, supporting various synchronization methods and emphasizing correctness and performance analysis. This paper details Crust's design and theoretical basis, showing how it bridges the gap between CRDT theory and practice, addressing tooling gaps to enhance adoption and real-world use. By enabling rigorous testing and performance evaluation, Crust has the potential to accelerate the development and adoption of CRDTs in real-world distributed systems.

INDEX TERMS CRDTs, conflict-free replicated data types, distributed systems, rust.

I. INTRODUCTION

Conflict-free Replicated Data Types (CRDTs) have emerged as essential components in contemporary distributed systems, providing a dependable method for data synchronization in the presence of network uncertainties and decentralized structures. By allowing concurrent updates across distributed nodes without the need for strict consensus, CRDTs ensure strong eventual consistency (SEC) while maintaining high availability [1]. Their significance has grown alongside the rising demand for fault-tolerant solutions in areas such as collaborative editing [2], [3], [4], databases [5] and the Internet of Things (IoT) [6]. Despite their solid theoretical foundation, the development and thorough evaluation of CRDTs—particularly in terms of correctness and performance—remains a challenging endeavor.

Although CRDT research has advanced significantly, with a majority of studies targeting algorithmic innovation and applications [1], [7], a notable gap remains: While significant progress has been made, the practical deployment of these algorithms is often hindered by the lack of standardized tools and methodologies for ensuring their correctness, performance, and also lacking a comprehensive framework in real-world scenarios. To back up this claim, an analysis of studies and GitHub repositories in Section II has been conducted. This reveals a fragmented CRDT ecosystem lacking modular frameworks for robust validation, performance evaluation, and security, hindering practical adoption. This fragmentation makes it challenging for developers to select the most appropriate CRDT for their needs, confidently validate its correctness, understand its performance characteristics, and integrate it securely into their distributed applications. The absence of a unified platform forces developers to piece together solutions from disparate sources, increasing complexity and the potential for errors. Therefore, the challenge is clear: create integrated frameworks to connect theoretical progress with dependable real-world use.

Addressing the aforementioned challenges and gaps, this paper presents the design and theoretical basis of the Crust framework, guided by the following research questions:

The associate editor coordinating the review of this manuscript and approving it for publication was Youngjin Kim.

1) Can a modular and extensible framework be designed and implemented in Rust to simplify and support the creation, validation, and performance analysis of CRDTs?
2) How can this framework provide comprehensive and configurable validation mechanisms for verifying CRDT correctness across diverse network conditions and consistency requirements?
3) What architecture and components are necessary to incorporate robust and consistent performance benchmarking capabilities for various CRDT implementations within such a framework?

Considering the complexity of the framework's architecture and emphasis on creating a solid base for CRDT development and evaluation, this paper focuses mainly on Crust's conceptual design and theoretical foundations, incorporating implementation specifics only as needed to clarify essential architectural concepts. The rest of this paper is organized as follows.

1) A brief review of CRDTs and CRDT landscape analysis by focus area are presented in Section II.
2) Mathematical foundations of synchronization in Crust are explained in Section III.
3) A novel framework Crust architecture is introduced and presented in Section IV, which includes the five key modules.
4) This is followed by Section V that concludes this article with some suggestions for further research.

II. RELATED WORK

A. OVERVIEW OF CRDTS

Conflict-free Replicated Data Types (CRDTs) have gained significance as a vital paradigm, providing a robust solution for achieving Strong Eventual Consistency (SEC) and ensuring availability in complex distributed environments. By enabling conflict-free data replication without the need for complex coordination protocols, as demonstrated by Shapiro et al. [1], CRDTs empower distributed nodes to independently update shared data and converge seamlessly toward a consistent state, even without centralized oversight. This feature is essential for maintaining both high availability and data consistency in contemporary distributed applications. As the demand for robust systems grows, the significance of CRDTs and the frameworks supporting their practical deployment becomes increasingly evident. Recognizing these core challenges and the potential of CRDTs inspires a comprehensive exploration of their underlying principles and current methodologies, which will be addressed in the subsequent sections.

CRDTs are designed to naturally deliver SEC, as outlined by Shapiro et al. [1], guaranteeing that all nodes will ultimately reach the same state. This is accomplished through mathematically reliable updates based on state, operations, or delta updates, as explained in Section III. Unlike standard eventual consistency, SEC ensures convergence even when faced with unpredictable network delays or partitions, providing stronger reliability.

CRDTs can be broadly classified according to their synchronization approaches and the types of data structures they are intended to handle. Depending on their synchronization mechanisms, CRDTs are divided into three main types, each presenting unique trade-offs regarding network usage, reliability needs, and complexity of implementation:

- State-based CRDTs (CvRDTs): State-based CRDTs, also known as Convergent Replicated Data Types, depend on sharing complete state information. This method ensures idempotent merges, which simplifies the conflict resolution process since merges are both repeatable and easy to execute. However, this advantage is offset by considerable network overhead, particularly in large-scale distributed systems where the size of the states can grow substantially [1].
- Operation-based CRDTs (CmRDTs): Operation-based CRDTs, also referred to as Commutative Replicated Data Types, enhance network efficiency by transmitting only the operations applied to the data. This approach lowers communication expenses but necessitates dependable delivery of operations to achieve convergence, often requiring more intricate operation designs to maintain commutativity [1].
- Delta-based CRDTs: Delta-based CRDTs adopt a hybrid model, sending only incremental state updates, or ''deltas'' between nodes. This approach seeks to strike a balance between the high network demands of state-based CRDTs and the strict reliability needs of operation-based CRDTs, providing a potentially more streamlined synchronization method [8].

In addition to synchronization mechanisms, CRDTs are designed as different data structures to address a wide range of data management requirements in distributed systems. The most common types include:

- Set-based CRDTs: Set-based CRDTs include CRDTs tailored for managing set-like data. Key examples are GSet (Grow-only Set), which permits only additions; 2P-Set (Two-Phase Set), which supports both additions and removals but prioritizes removal operations; and ORSet (Observed-Removed Set), which more adaptable handles concurrent additions and removals by maintaining causal relationships [9].
- Counter-based CRDTs: Counter-based CRDTs are developed to manage distributed counters that can be simultaneously incremented and decremented across multiple nodes. GCounter (Grow-only Counter) is limited to increment operations, whereas PNCounter (Positive-Negative Counter) supports both increments and decrements, ensuring consistency across distributed nodes [9].
- Register-based CRDTs: Register-based CRDTs are designed to handle distributed registers. The LWWRegister (Last-Write-Wins Register) addresses conflicts by

choosing the most recent write according to timestamps, whereas the MVRegister (Multi-Value Register) can store multiple concurrent values, necessitating a method to either resolve or display these values to the application [9].

- Map-based CRDTs: Map-based CRDTs are utilized in distributed key-value stores, enabling simultaneous updates to key-value pairs across nodes. The ORMap (Observed-Remove Map) leverages observed-remove semantics [10], supporting concurrent updates and removals while preserving consistency through causality tracking. More sophisticated map CRDTs can be constructed by embedding other CRDTs, like sets or counters, as values within the map, accommodating various data structures and conflict resolution requirements for the values.
- Sequence-based CRDTs: Sequence-based CRDTs, employed in applications like collaborative text editing, handle ordered sequences of data, facilitating operations such as insertion and deletion while ensuring consistent ordering across distributed environments. Examples include Logoot [11], RGA (Replicated Growable Array) [12], and Yjs (a widely used JavaScript CRDT library featuring a sequence CRDT) [3].

B. CRDT LANDSCAPE ANALYSIS BY FOCUS AREA

Understanding the variety of CRDT types outlined in the prior section is essential for recognizing their potential uses in distributed systems. However, the practical adoption and effective utilization of these CRDTs are not only on their theoretical foundations. To achieve a comprehensive quantitative understanding of the current landscape of CRDT development and identify areas requiring enhancement, a dual approach analysis was conducted, exploring both academic research publications and openly accessible CRDT implementations.

For the research publications analysis, an extensive search on Google Scholar using the keywords ''Conflict-free replicated data type'' and ''CRDT'' for publications released after 2011 were conducted. These terms are the standard terminology in the CRDT research, and the post-2011 period was selected to reflect the field's development following the formal introduction of CRDTs by Shapiro et al. [1]. The initial search returned around 1,130 publications. To refine this dataset, inaccessible entries or broken links were excluded, resulting in a final collection of 902 publications. Each abstract within this collection was then manually reviewed and categorized into one of six key focus areas: Theoretical Foundations, Algorithmic Design and Innovation, Application and Use Case Exploration, Validation and Correctness Verification, Performance Benchmarking and Evaluation, and Security. Recognizing the subjectivity and potential for misclassification in quick, abstract-only reviews, each paper was assigned to the focus area that best captured its primary contribution. For instance, papers mainly introducing new CRDT algorithms were categorized under ''Algorithmic Design and Innovation'', even if they briefly mentioned applications, validation or benchmarking.

Breakdown of Research Publication Focus Areas (Percentages derived from 902 Total Publications):

- Theoretical Foundations: 3.1% (28 publications)
- Algorithmic Design and Innovation: 34.3% (309 publications)
- Application and Use Case Exploration: 54.5% (492 publications)
- Validation and Correctness Verification: 4.8% (43 publications)
- Performance Benchmarking and Evaluation: 1.1% (10 publications)
- Security: 2.2% (20 publications)

The breakdown of research focus areas shows a strong concentration on Application and Use Case Exploration (54.5%) and Algorithmic Design and Innovation (34.3%), indicating a field deeply engaged in investigating CRDT applications across various domains and consistently creating new CRDT variants. For instance, it provides a typical example of application-focused research, exploring CRDTs in database management at Edge [13]. Similarly, in [14], it exemplifies the algorithmic innovation trend, proposing a novel CRDT structure for reversible CRDTs. However, significantly lower percentages are seen in areas vital for practical implementation and reliable system development: Validation and Correctness Verification (4.8%) and Performance Benchmarking and Evaluation (a mere 1.1%). For instance, while [15] contributes to validation efforts, such studies remain relatively scarce. Likewise, it is a rare example of performance benchmarking, showing in the [16]. This marked under-representation in validation and benchmarking studies points to a clear deficiency in the thorough assessment of CRDT implementation quality and performance, reinforcing the need for frameworks like Crust that emphasize these elements. The relatively low percentage for Security (2.2%) also highlights the need for increased focus on security aspects within the CRDT field, aligning with the wider challenges in distributed system security. A typical example of security-focused research is [17], which addresses integrity, authentication and confidentiality in CRDTs. Finally, for theoretical foundations, a typical example could be represented by [1] which has been mentioned many time in this paper as the fundamental work for CRDT.

For the evaluations of existing CRDT implementations, a search on GitHub using the same keywords ''Conflict-free replicated data type'' and ''CRDT'' and the same date range (after 2011) was conducted. To handle the large number of repositories and focus on those with some degree of community engagement, a filter for repositories with more than one star was applied. Although star count is an imperfect metric and does not directly reflect credibility, it provides a practical way to identify projects with minimal community awareness and adoption, improving the chances of finding

relevant and possibly more mature implementations. The initial search produced approximately 690 repositories. After filtering out those that were not code-based implementations, lacked sufficient documentation, or served mainly as tutorials, a final set of 568 repositories was derived. These repositories were then manually categorized based on a review of repository names, descriptions, and README files into twelve categories reflecting their primary focus: Framework, Library, Showcase and Examples, Database Focus, Collaborative Editing Focus, Mobile and Edge Computing Focus, P2P Focus, Blockchain Focus, Other Use Case Focus, Benchmark, Correctness, and Security and Privacy.

It is essential to clearly define the term ''Framework'' and ''Library'' categories, as these form the basis of the analysis. A ''Library'' was defined as a set of CRDT data structures code in a specific programming language or adaptations of a widely recognized library, such as Yjs [3]. These libraries were mainly designed to supply developers with implementations of different CRDTs, allowing their integration into various applications.

In contrast, a ''Framework'' was described as a broader, more robust tool aimed at supporting the development, validation, and performance assessment of CRDTs. This definition, aligned with the intended purpose of Crust, requires a framework to provide not only CRDT implementations but also offer features for verifying correctness, conducting performance benchmarks, and providing extensive development assistance.

Due to the subjective nature of categorization based on limited public information in repositories, repositories were placed into the category that best represented their primary focus, consistent with the approach used in the analysis of research publications. Analysis of GitHub Repository Focus Areas (Percentages based on 568 Total Repositories):

- Framework: 0% (0 repositories)
- Library: 40.5% (230 repositories)
- Showcase/Examples: 6.0% (34 repositories)
- Other Use Case Focus: 10.7% (61 repositories)
- Database Focus: 14.4% (82 repositories)
- Collaborative Editing Focus: 19.5% (111 repositories)
- Mobile/Edge Computing Focus: 0.7% (4 repositories)
- P2P Focus: 0.9% (5 repositories)
- Blockchain Focus: 3.3% (19 repositories)
- Benchmark: 0.9% (5 repositories)
- Correctness: 1.4% (8 repositories)
- Security/Privacy: 1.6% (9 repositories)

The evaluation of GitHub repositories highlights a prevalence of Libraries (40.5%), indicating the typical approach involves offering sets of CRDT data structures. For example, it exemplifies this trend, providing a Rust library with multiple implementations in [18]. A significant share of repositories is also application-focused, with those focused on Collaborative Editing (19.5%) and Database (14.4%) standing out. In [19], it demonstrates a typical collaborative editing focused repository. Similarly, it showcases a database focused implementation, integrating CRDTs for local-first, NoSQL-database [20]. When combining all Focus categories as the Application and Use Case Exploration in the Research Publication Focus Areas, the portion rises significantly (49.6%). However, under the refined definition, no repositories qualified as dedicated ''Frameworks''. Furthermore, repositories explicitly focused on Benchmark (0.9%) and Correctness (1.4%) are extremely rare. It is a rare example of a benchmarking repository in [21]. Likewise, it is one of the few repositories dedicated to correctness validation [7]. This marked scarcity of CRDT frameworks, alongside the limited presence of tools devoted to benchmarking and correctness validation in open-source repositories, directly underscores the identified shortfall in practical tooling and strengthens the case for a comprehensive framework solution. The minimal representation of Security/Privacy-focused repositories (1.6%), consistent with trends in research publications, further highlights the relative neglect of security considerations in current CRDT implementations. It represents a security focused repository by addressing byzantine fault tolerance [22].

This two-part quantitative analysis, which looks at both academic research trends and open-source projects, strongly supports the need for the Crust framework. The research publication review shows a lack of focus on validation and performance testing, while the implementation study uncovers a significant shortage of dedicated CRDT frameworks and tools for benchmarking and correctness. These findings provide solid evidence for the paper's main argument: there's an urgent need for a flexible, customizable, and all-in-one framework like Crust to simplify CRDT development, enable thorough testing, and promote their practical use, connecting theoretical progress with reliable, real-world applications.

III. MATHEMATICAL FOUNDATIONS OF SYNCHRONIZATION IN CRUST

Before exploring the detailed design of the Crust framework modules, it is important to revisit and formalize the theoretical foundations that support CRDTs and shape the framework's architecture. This section offers a brief summary of the mathematical proofs and requirements that govern state-based, operation-based, and delta-based CRDTs [1], [8], [9], emphasizing the core principles on which Crust is constructed. This theoretical foundation establishes a solid framework for comprehending the essential characteristics of CRDTs and how Crust ensures their accurate implementation and verification.

A. MATHEMATICAL FOUNDATIONS OF STATE-BASED CRDTS

State-based CRDTs ensure convergence by sharing and merging the full state of nodes.

Replica Set $R = \{r_1, \ldots, r_n\}$: Finite set of nodes, each replica $r \in R$ has initial state $s_0^r \in S$

State Space $(S, \sqsubseteq, \sqcup)$: Set of all possible states for a replica S, partial order $\sqsubseteq$, least Upper Bound (LUB) operator $\sqcup$

Local Update Function (U): Set of local updates U, local update mutate the state $u \in U : S \to S$, in which $s^r \sqsubseteq u(s^r)$

Merge Function ($\bigsqcup : S \times S \to S$): Computes the LUB of two states

Properties of the Merge Function (Join-Semilattice and Monotonicity):

- Associativity: $(s^{r_1} \sqcup s^{r_2}) \sqcup s^{r_3} = s^{r_1} \sqcup (s^{r_2} \sqcup s^{r_3})$
- Commutativity: $s^{r_1} \sqcup s^{r_2} = s^{r_2} \sqcup s^{r_1}$
- Idempotence: $s^r \sqcup s^r = s^r$
- Monotonicity: $s^r \sqsubseteq s^r \sqcup s^r$

B. MATHEMATICAL FOUNDATIONS OF OPERATION-BASED CRDTS

Operation-based CRDTs ensure convergence by broadcasting and causally applying operations.

Replica Set $R = \{r_1, \ldots, r_n\}$: Finite set of nodes, Each replica $r \in R$ has initial state $s_0^r \in S$

State Space $(S, \sqsubseteq)$: Set of all possible states for a replica S, partial order $\sqsubseteq$

Local Update Function (U): Set of local updates U, local update mutate the state $u \in U : S \to S$, in which $s^r \sqsubseteq u(s^r)$

Prepare-update (Operation Generation Function): Set of operations generate methods P, generates operation message m from source replica $p : S \to \text{Msg}$

Effect-update (Apply Function): Set of operation apply methods E, Applies operation m to state $e : S \times \text{Msg} \to S$

Properties of the Apply Function:

- Causal order: If $m_1 \to m_2$, then m_1 is delivered before m_2 at all nodes
- Commutativity: $e_i, e_j \in E$: $s \circ e_i \circ e_j \equiv s \circ e_j \circ e_i$

C. MATHEMATICAL FOUNDATIONS OF DELTA-BASED CRDTS

Delta-based CRDTs propagate partial state changes (deltas) and merge them for efficient state synchronization.

Replica Set $R = \{r_1, \ldots, r_n\}$: Finite set of nodes, Each replica $r \in R$ has initial state $s_0^r \in S$

State Space $(S, \sqsubseteq, \sqcup)$: Set of all possible states for a replica S, Partial order $\sqsubseteq$, Least Upper Bound (LUB) operator $\sqcup$

Local Update Function (U): Set of local updates U, local update mutate the state $u \in U : S \to \Delta$, in which $s^r \sqsubseteq u(s^r)$, you might see it as delta mutator m^δ in other report

Delta Set (Δ): Set of all possible deltas, representing partial state changes, $\Delta \sqsubseteq S$

Delta Group ($D = \bigsqcup\{\delta_1, \delta_2, \ldots\}$): Join of multiple deltas, transmitted as a single message

Causal Context (C): Track causal dependencies to avoid redundant delta transmission $C \sqsubseteq \mathbb{N} \times \mathbb{R}$

Delta Generating Function (F^δ): Set of deltas generate methods F^δ, generates operation message m from source replica $f : S \to \delta$

Delta Merge Function ($\bigsqcup^\delta : S \times D \to S$): Computes the LUB of State and Delta Group

Properties of Delta Merge Function (Join-Semilattice and Monotonicity): Same as in Section III-A, but in Delta-based CRDT, the $\sqcup$ works uniformly on: State + State ($s^{r_1} \sqcup s^{r_2}$), State + Delta ($s^r \sqcup \delta$), Delta + Delta ($\delta_1 \sqcup \delta_2$)

D. MATHEMATICAL REQUIREMENTS FOR CRUST

Following the detailed mathematical definition of different synchronization mechanisms of CRDTs, I can now leverage these insights to establish a clear set of mathematical requirements for the development of Crust.

To implement State-based CRDTs in Crust:

- A method representing the state of each replica
- A method performing local updates on the state
- A merge method takes two states and combines them into a new state, ensuring the merge operation satisfies associativity, commutativity, and idempotence

To implement Operation-based CRDTs in Crust:

- A method representing the state of each replica
- A method performing local updates on the state
- A method generating operation messages corresponding to local updates
- A method applying operation messages to a replica's state, ensuring that operations are applied in causal order and commutativity

To implement Delta-based CRDTs in Crust:

- A method representing the state of each replica
- A method performing local updates on the state
- A method generating delta messages representing partial state changes resulting from local updates
- A method representing and potentially grouping multiple delta messages
- A mechanism tracking causal context for efficient delta propagation
- A merge method combining a state with a delta, a state with a group of deltas, or two deltas (or groups of deltas) into a new state or delta, ensuring the merge operation aligns with associativity, commutativity, and idempotence

IV. CRUST FRAMEWORK ARCHITECTURE

Crust adopts a modular architecture, organized into five primary modules to maintain a distinct separation of concerns, enhancing maintainability, extensibility, and testability. These modules are Core, Config, Network, Validation, and Benchmark. The design and implementation of Crust aim to tackle the shortcomings identified in existing CRDT frameworks in Section II while ensuring compliance with the strict convergence and consistency principles established by CRDT theory in Section III. The Crust framework architecture can be seen in Fig 1.

- The Core Module forms the fundation of Crust, responsible for management of CRDT data structures, their operations, and synchronization settings.
- The Config Module is responsible for framework-wide configuration, such as Kubernetes setups, enabling developers to tailor Crust's behavior and deployment to their needs.

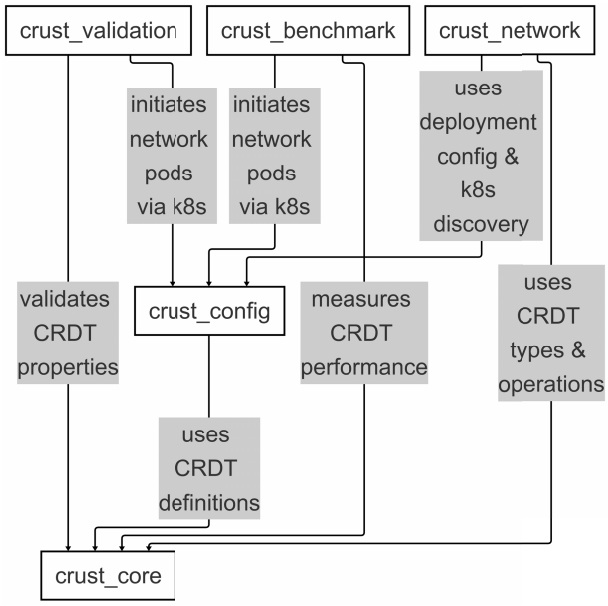

FIGURE 1. Crust framework architecture.

- The Network Module manages all network-related communication between Crust instances, including message sending and receiving.
- The Validation Module offers a robust system for checking the accuracy and reliability of CRDT implementations, ensuring compliance with CRDT properties.
- The Benchmark Module is designed to support the performance evaluation and benchmarking of CRDT implementations within Crust framework.

A. DESIGN OF THE CORE MODULE

The Core Module is the central component of Crust, responsible for the lifecycle management of CRDTs in Fig 2. It is designed to be highly flexible and extensible, supporting a range of CRDT data structures and synchronization mechanisms.

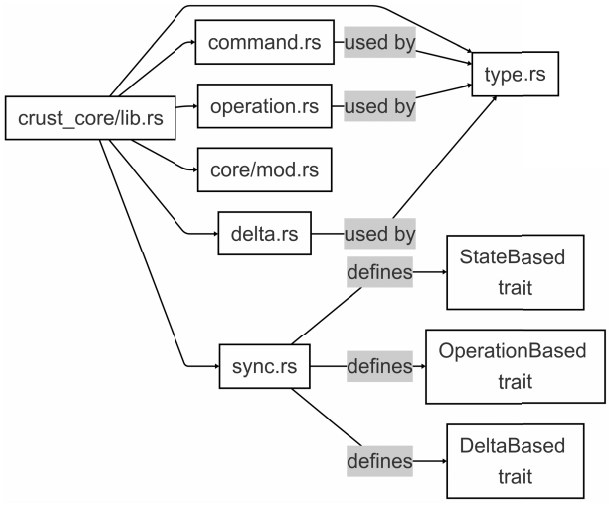

FIGURE 2. Core module.

The module's main responsibilities include:

- Top-level Modules (`lib.rs`): The Core Module is logically organized into sub-modules within the `lib.rs` file, promoting modularity and maintainability. These sub-modules are:
  - `command`: Defines atomic mutation operations for CRDT data structures.
  - `core`: Contains concrete implementations of CRDT data structures.
  - `delta`: Handles delta propagation and aggregation logic.
  - `operation`: Manages operation-based synchronization semantics.
  - `sync`: Implements synchronization mechanisms and configuration.
  - `type`: Provides CRDT instance management and buffers.
- Command Module (command.rs): The `command` sub-module defines the `InnerCommand<K>` enum, which serves as the fundamental building block for CRDT mutations as Local Update Function which I mentioned in Section III. Crust recognizes this underlying similarity and abstracts these stages into a more generalized concept key components include:
  - `CounterIncrement`/ `CounterDecrement`
  - `SetAdd`/`SetRemove`
  - `TextInsert`/`TextDelete`
- Core Module (`core/`): The `core` directory contains type-specific CRDT implementations. The CRDT implementations implement the traits defined by `sync.rs`.
- Delta Module (`delta.rs`): The `delta` sub-module manages Delta-based CRDT synchronization through:
  - `CrdtDelta<K>` enum: Wrapper for delta payloads
- Operation Module (`operation.rs`): The `operation` sub-module defines CRDT operation semantics:
  - `CrdtOperation<K>` enum: Unified operation container
  - Operation batching support through `Vec<CrdtOperation<K\gg`
- Sync Module (`sync.rs`): The `sync` sub-module orchestrates synchronization mechanisms:
  - `SyncConfig` struct: Central configuration hub
    - `sync_type`: Enum (State/Operation/Delta)
    - `sync_mode`: Enum $Immediate/BatchTime/ BatchCount)
    - `batch_times`: Threshold for count-based batching
    - `batching_interval`: Duration for time-based batching

    - Notably, the `sync_mode` enum $Immediate/BatchTime/BatchCount) is inspired by the 'delta-group' concept from delta-based CRDTs, as explored in Section III.
  - Core traits:
    - `StateBased`: Defines `merge()` for State-based CRDTs
    - `OperationBased`: Provides `apply()` for Operation-based CRDTs
    - `DeltaBased`: Manages `generate_delta()` and `merge_delta()` for Delta-based CRDTs
- Type Module (`type.rs`): The `type` sub-module manages CRDT instances:
  - `CrdtTypeVariant<K>` enum: Container for CRDT variants
  - `CrdtType<K>` struct: Main CRDT instance wrapper. It contains not only the methods in the traits defined by `sync.rs` that I just mentioned above, but also includes methods such as:
    - `operations_buffer`: Batches unprocessed operations
    - `deltas_buffer`: Accumulates pending deltas
    - `get_state()`: Exposes current CRDT state as JSON
    - `apply_command()`: Translates commands to operations
    - `generate_operation_count_based()`: count based synchronization triggers
    - `generate_delta_time_based()`: time based synchronization triggers

The extensive use of enums, such as `InnerCommand`, `CrdtDelta`, `CrdtOperation`, and `CrdtTypeVariant` throughout the Core Module is a deliberate design choice. Enums provide a type-safe and extensible mechanism in Rust to represent a closed set of possible states or variations within each module. For instance, the `InnerCommand` enum enumerates all possible mutation operations supported by the framework; the `CrdtTypeVariant` enumerates all the possible data structures implemented by the framework.

The separation of the `command` and `operation` modules is intentional. While the `command` module defines user-facing, high-level internal mutation intents, the `operation` module deals with the external representation of these mutations as `CrdtOperation` specifically used for Operation-based CRDTs. This decoupling allows for flexibility in how commands are translated into states, operations or deltas and provides a unified operation format for external synchronization and internal processing.

The Sync Module utilizes traits like `StateBased`, `OperationBased`, and `DeltaBased` to define synchronization mechanisms. This trait-based approach enables polymorphism, allowing different CRDT data structures to implement the most appropriate synchronization mechanism while adhering to a consistent interface for the Sync Module to interact with. This design promotes extensibility and adaptability to various CRDT synchronization paradigms.

B. DESIGN OF THE CONFIG MODULE

The Config Module in Crust is responsible for managing the framework-level configuration, with a special emphasis on the deployment and operational environment within Kubernetes in Fig 3. It simplifies the complexities of Kubernetes resource management and provides a programmatic interface for setting up and deploying Crust instances. The key design considerations and components of the Config Module are:

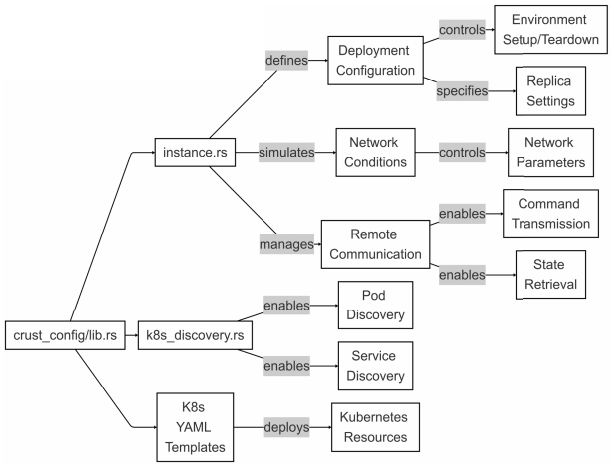

FIGURE 3. Config module.

- Top-level Modules (`lib.rs`): The Config Module is logically organized into sub-modules within the `lib.rs` file, promoting modularity and maintainability. These sub-modules are:
  - `instance`: Manages Kubernetes deployment lifecycle and network configurations
  - `k8s_discovery`: Manages Kubernetes cluster discovery and pod coordination
- Instance Module (`instance.rs`): The core configuration component handling cluster orchestration:
  - `DeploymentConfig` struct: Central configuration container
    - `num_replicas`: Number of cluster replicas
    - `crdt_type`: CRDT data structures
    - `sync_type`: Synchronization mechanisms $State/Operation/Delta)
    - `sync_mode`: Sync triggering mechanism (Immediate/Batch)
    - `network_scenario`: Network condition simulations
  - `NetworkScenario` struct: Defines network characteristics
    - `packet_loss`: Simulated packet loss probability

    - `latency`: Artificial network delay in milliseconds
    - `bandwidth`: Throttled network capacity in bytes/sec
- Kubernetes Discovery Module (`k8s_discovery.rs`): Manages cluster node discovery and coordination:
  - `get_replica_pod_names()`: Discovers active cluster pods using Kubernetes API
- Kubernetes Integration (`k8s/`): Manages cluster lifecycle through YAML templates:
  - `deployment.yaml`: CRDT node deployment specification
  - `service.yaml`: Network service definition
  - `service_account.yaml`: RBAC permissions
- Cluster Operations: Implements essential Kubernetes CRUD operations
  - `setup_remote_test_environment()`
  - `teardown_remote_test_environment()`
- Network Configuration: Dynamic network parameter adjustment
  - `update_packet_loss()`: Applies packet loss patches
  - `update_latency()`: Injects artificial delays
  - `update_bandwidth()`: Throttles network throughput
  - `update_replicas()`: Scales deployment replicas
- Command Propagation: Network-aware CRDT operation handling
  - `send_command_to_instance()`: Reliable operation delivery
  - `send_command_to_instance_with_loss()`: Simulates message loss
  - `get_state_from_instance()`: State snapshot retrieval

The Config Module is intentionally structured with separate `instance` and `k8s_discovery` sub-modules to promote a clear separation of concerns. The `instance` module focuses on the lifecycle management of Crust deployments within Kubernetes, encompassing configuration and orchestration, while the `k8s_discovery` module is dedicated to handling Kubernetes cluster awareness and dynamic pod discovery. This division enhances modularity and maintainability by isolating distinct functionalities.

A key strength and potential novelty of the Config Module lies in its dynamic network and deployment reconfiguration capabilities. While the module leverages YAML templates for initial Kubernetes deployment specifications, it goes beyond static configuration by providing programmatic functions like `update_packet_loss()`, `update_latency()`, `update_bandwidth()`, and `update_replicas()`. This dynamic adjustment mechanism allows for runtime modification of critical deployment parameters, such as network conditions and cluster size, without requiring redeployment from YAML. This feature is particularly crucial for facilitating comprehensive validation, benchmarking, and experimentation within diverse and dynamically changing environments, enabling rigorous evaluation of the different CRDTs.

C. DESIGN OF THE NETWORK MODULE

The Network Module in Crust is responsible for all communication aspects between distributed Crust instances in Fig 4. It is designed to be Kubernetes-aware, facilitating message exchange, service discovery, and handling various network communication patterns required for CRDT synchronization. The module's key design features and components are:

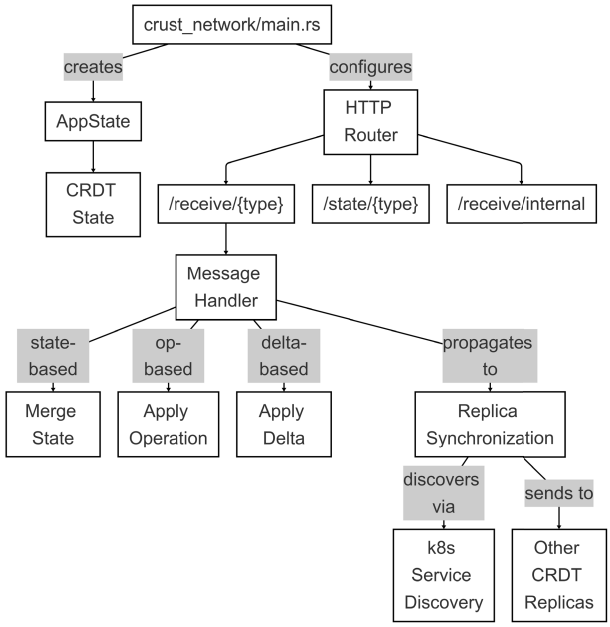

FIGURE 4. Network module.

- Top-level Modules (`main.rs`): `main.rs` at the module root since this is a binary module. The sub-module includes:
  - `message`: Defines network message formats for CRDT synchronization
  - `receiver`: Handles incoming message processing and state management
  - `sender`: Implements message broadcasting and network transmission
- Message Module (`message.rs`): Defines the unified network message format:
  - `NetworkMessage<K>` enum: Encapsulates all CRDT synchronization payloads
    - `State` variant: Contains full CRDT state snapshots

    - * `Operation` variant: Carries CRDT operations with sender metadata
    - * `Delta` variant: Transmits delta updates
- Receiver Module (`receiver.rs`): Implements message processing and state synchronization:
  - – `AppState` struct: Manages node-specific CRDT instances
    - * Thread-safe RwLock synchronization for concurrent access
    - * Dynamic CRDT data structure initialization that provided by `CrdtType`
    - * State snapshot generation via `get_state()`
  - – Message handlers:
    - * `receive_message_from_other_instances()`: Processes external node messages
    - * `receive_message_from_internal()`: Handles internal API commands
    - * Implements conflict avoidance through sender ID checks
- Sender Module (`sender.rs`): Implements message propagation:
  - – `NetworkSender` struct: Manages outbound communication
    - * `broadcast_message()`: Broadcast CRDT messages to all cluster nodes
    - * Loopback prevention through sender ID filtering
- Containerization (`DockerFile`): Deployment docker container for Kubernetes

The Network Module adopts a design with distinct `sender` and `receiver` sub-modules to achieve a clear separation of network responsibilities. The `sender` module is dedicated to handling outbound message transmission and broadcasting, while the `receiver` module focuses on processing inbound messages and updating local application state. This separation simplifies the internal logic of each sub-module, enhances code organization, and improves maintainability by isolating network sending and receiving concerns.

The `NetworkMessage` enum is employed as the unified format for all network communication within Crust. This enum-based approach provides type safety and clarity by explicitly defining the possible types of messages exchanged: `State`,`Operation`, and `Delta`. Using an enum ensures that message handling logic is comprehensive and accounts for all defined message types, while also allowing for potential extension with new message variants in the future in a type-safe manner.

D. DESIGN OF THE VALIDATION MODULE

The Validation Module in Crust is a critical component, focusing on thoroughly testing and confirming the accuracy and resilience of CRDT implementations within the framework, as illustrated 5. It aims to ensure adherence to the theoretical convergence and consistency guarantees inherent to CRDTs, while also assessing correctness and durability across diverse network scenarios. The module is divided into two main categories: Local Validation and Remote Validation, further organized by CRDT synchronization mechanisms (State-based, Operation-based, Delta-based) and specific validation properties.

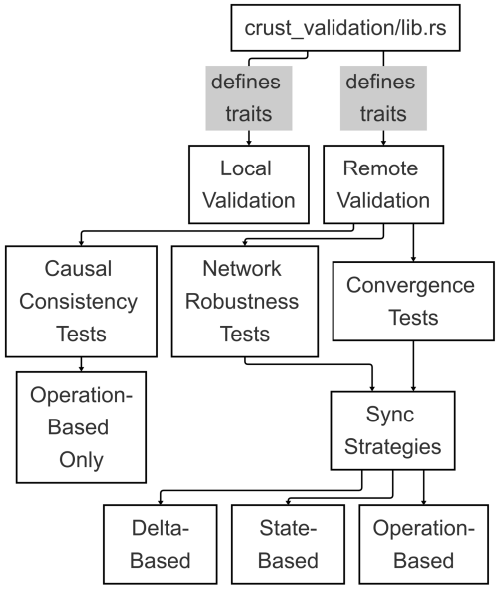

FIGURE 5. Validation module.

- Top-level Modules (`lib.rs`): The Validation Module is logically organized into sub-modules within the `lib.rs` file, promoting modularity and maintainability. These sub-modules are:
  - – `local_tests`: Single-node test implementations
  - – `local_validation`: Verifies CRDT properties through formal verification
  - – `remote_tests`: Cluster-level test scenarios
  - – `remote_validation`: Tests distributed convergence and fault tolerance
- Local Validation Module (`local_validation.rs`): Implements axiomatic verification of CRDT properties:
  - – `StateBasedValidation` Trait
    - * `state_associativity()`: Tests if merging CRDT states is associative, meaning the order of merging multiple states doesn't affect the final merged state
    - * `state_commutativity()`: Tests if merging CRDT states is commutative, meaning merging states in any order results in the same state
    - * `state_idempotence()`: Tests if merging a CRDT state with itself multiple times is equivalent to merging it just once, ensuring no cumulative effect from redundant merges

    * `state_monotonicity()`:Tests if merging a CRDT state with an updated version of itself always results in a state that reflects the updates, ensuring that information is never lost or reverted during merges
  - `OperationBasedValidation` Trait
    * `operation_commutativity()`: Tests if applying operations in different orders leads to the same final CRDT state, regardless of operation sequence
    * `operation_delivery_precondition()`: Tests if operations are only applied when their necessary preconditions are met, ensuring that operations are executed in a valid context
    * `operation_effect_relation()`: Tests the correctness of the effects of operations on the CRDT state, verifying that applying an operation leads to the expected state changes
  - `DeltaBasedValidation` Trait
    * `delta_associativity()`: Tests if merging CRDT deltas is associative, meaning the order of merging multiple deltas doesn't affect the final merged delta
    * `delta_commutativity()`: Tests if applying delta mutations in different orders results in the same final state, crucial for ensuring correct merging of deltas from different sources
    * `delta_idempotence()`: Tests if applying the same delta multiple times is equivalent to applying it once, ensuring efficient delta processing without redundant state changes
    * `delta_state_composability()`: Tests if applying a delta to a CRDT state correctly reflects the changes represented by the delta
- Remote Validation Module $`remote_validation.rs`): Implements distributed system validation:
  - Convergence Validation
    * `{state, operation, delta}_converge_concurrent_operations()`: Validates convergence when multiple replicas concurrently perform operations and then synchronize, testing the CRDT's ability to handle parallel updates
    * `{state, operation, delta}_converge_delayed_deliveries()`: Validates convergence in the presence of network delays, where messages might arrive out of order or with significant latency, simulating realistic network conditions
    * `{state, operation, delta}_converge_mixed_operations()`:Tests convergence under a mixed workload of concurrent operations and delayed message deliveries
    * `{state, operation, delta}_converge_under_load()`: Evaluates convergence when the system is subjected to a high operation load, assessing if convergence is maintained even under stress and performance pressure
  - Causal Consistency Validation
    * `operation_causal_order_simple_dependency()`: Tests if operations with a straightforward causal dependency are processed and reflected in the correct causal order across replicas
    * `operation_causal_order_complex_dependency()`: Tests causal order preservation with more intricate dependency chains or trees of operations, ensuring correctness in complex operation sequences
    * `operation_causal_order_concurrent_dependency()`: Validates causal consistency when there are concurrent operations that might appear dependent but are actually causally independent, ensuring no false dependency violations
    * `operation_causal_order_delayed_delivery()`: Tests if causal order is maintained even when network delays and out-of-order message delivery occur, challenging the consistency mechanisms under network latency
    * State-based CRDTs merge entire states. The causal history of individual operations is less important than the final merged state. Thus, these tests are less relevant for state-based CRDTs. Similarly, Delta-based CRDTs are essentially an optimization of state-based CRDTs that transmit only deltas rather than the full state. They inherit the state-based property of not needing to track explicit causal ordering dependencies.
  - Network Robustness Validation
    * `{state, operation, delta}_robustness_message_loss()`: Tests the CRDT's ability to achieve convergence and maintain consistency despite message loss in the network, simulating unreliable network conditions
    * `{operation, delta}_robustness_message_reordering()`: Tests the CRDT's handling of messages that arrive in a different order than they were sent, due to network reordering, ensuring resilience to packet delivery order variations. State-based CRDTs merge entire states. The order of state messages is less critical than the final merged state. Therefore, message reordering tests are less critical.
    * `{state, operation, delta}_robustness_network_partition()`:

Evaluates how the CRDT behaves during and after a network partition, where parts of the network become disconnected, and validates its recovery and convergence once the partition is resolved, testing fault tolerance

Local Validation focuses on unit-level axiomatic verification of fundamental CRDT properties like associativity, commutativity, and idempotence. These tests are typically performed locally. In contrast, Remote Validation targets the distributed system behavior of Crust, examining convergence, causal consistency, and network robustness in cluster-level deployments. This two-pronged approach allows for comprehensive validation, starting with verifying core CRDT properties locally and then extending to system-level behavior in distributed scenarios while adhere to the formal mathematical principles that defined by Section III.

E. DESIGN OF THE BENCHMARK MODULE

The Benchmark Module in Crust is designed to systematically evaluate the performance and resource utilization of Crust under various configurations and workloads in Fig 6. It provides a framework for automated benchmarking, allowing for in-depth analysis of different CRDT data structures, synchronization mechanisms, synchronization modes, and workload patterns. The module is structured into several key components, each with a distinct responsibility in the benchmarking process.

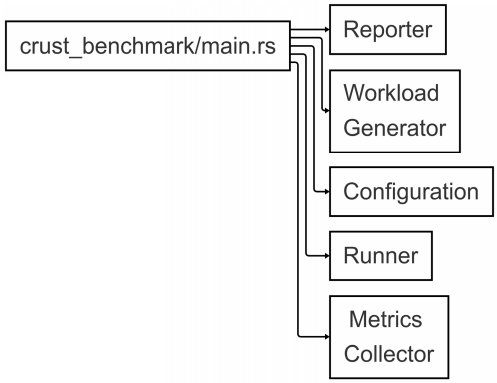

FIGURE 6. Benchmark module.

- Top-level Modules (`main.rs`): `main.rs` at the module root since this is a binary module. The sub-module includes:
  - `collector`: Responsible for collecting metrics during benchmark execution.
  - `config`: Manages the configuration parameters for benchmarks.
  - `metrics`: Defines the structure and types of metrics to be collected and reported.
  - `reporter`: Generates benchmark reports based on collected metrics.
  - `runner`: Orchestrates the entire benchmark execution workflow.
  - `workload`: Defines and generates the workload to be applied during benchmarking.
- Collector Module $`collector/metrics_collector.rs`): Implements metrics aggregation pipeline:
  - `MetricsCollector` struct
    - `define_metrics()`: Predefine the suite of metrics that will be collected during the benchmark that are defined in metrics module
    - `start_collecting()`: Initializes metric buffers
    - `stop_collecting()`: Finalizes data collection
  - Supported Metric Categories:
    - Performance: Latency, Throughput, Duration
    - Consistency: Convergence Time
    - Resource: CPU/Memory Usage
    - Error: Error Rates
- Config Module $`config/benchmark_config.rs`): Central benchmark configuration:
  - `BenchmarkConfig` struct
    - `crdt_type`: CRDT implementation variant
    - `sync_type`: State/Operation/Delta synchronization
    - `sync_mode`: Immediate/Batch synchronization
    - `workload_config`: Workload parameters
- Metrics Module $`metrics/metrics_definition.rs`): Defines measurement ontology:
  - `MetricType` enum: Enumerates the core types of metrics that can be measured. This includes `Latency`, `Throughput`, `CpuUsage`, `MemoryUsage`, `ErrorRate`, `ConvergenceTime`, `Size`, and `Duration`.
  - `MetricCategory` enum: Defines the broad measurement domains to which metrics belong. Categories like `Performance`, `Consistency`, `Resource`, and `Error` help in organizing and interpreting the collected data within meaningful contexts.
  - `MetricAggregation` enum: Lists the statistical operations that can be applied to collected metric data. Options like `Average`, `P50`, `P90`, `P99`, `Min`, `Max`, `Total`, and `Value` offer flexibility in summarizing and analyzing the raw metric values.
  - `MetricDataType` enum: Specifies the data formats for metrics, such as `Numeric`, `TimeDuration`, `Percentage`, `Integer`, and `Text`. This ensures that metrics are handled and reported with appropriate data representations.
  - `Metric` struct: This struct is the fundamental unit for defining a specific metric to be collected. It combines a `MetricType`, `MetricCategory`, a list of `MetricAggregations`, and a `MetricData`

`Type`. By combining these enums, the `Metric` struct provides a blueprint for each measurement, allowing for precise and configurable metric definitions throughout the benchmark system.

- Reporter Module `$reporter/benchmark_report.rs`): Generates analysis outputs:
  - `BenchmarkReport` struct
    - `generate_report()`: Produces consolidated results
    - Statistical calculations: Calculate `Average`, `P50`, `P90`, `P99`, `Min`, `Max`, `Total`
  - Output Formats:
    - Prometheus-compatible metrics: For integration with monitoring systems like Prometheus, allowing real-time monitoring and alerting based on benchmark results in operational environments.
- Runner Module `$runner/benchmark_runner.rs`):
  Manages benchmark lifecycle:
  - `BenchmarkRunner` struct
    - `run_benchmark()`: Combine all the structs from the other modules and defines execution flows
  - Error Handling:
    - `BenchmarkError` enum: Defines distinct categories of benchmark errors, such as `KubernetesError`, `CollectorError`, `WorkloadError`, and `ReporterError`. This categorization helps in pinpointing the source of failures.
    - Graceful teardown procedures: In case of errors, the runner is designed to execute graceful teardown procedures, ensuring that resources are cleaned up and the system returns to a stable state, even if the benchmark run is not fully successful.
- Workload Module (`workload/`): Generates operational stress:
  - `WorkloadConfig` struct
    - `command_mix`: Utilizes `CommandMixConfig` to specify the operation distribution profiles. This configuration allows users to define the mix of different CRDT operations (e.g., adds, removes, updates) within the workload, and their respective ratios, to simulate different usage patterns.
    - `num_commands`: Optionally specifies the load intensity by setting the total number of commands to be generated during the benchmark. This allows control over the duration and scale of the workload applied.
  - `WorkloadGenerator` struct
    - `generate_inner_command_workload()`: This function is critical for generating CRDT-specific operations based on the configured `crdt_type` and `command_mix`. It intelligently creates workloads tailored to the specific CRDT being tested (Counter, Graph, Set, Text), ensuring relevant and realistic operational stress.

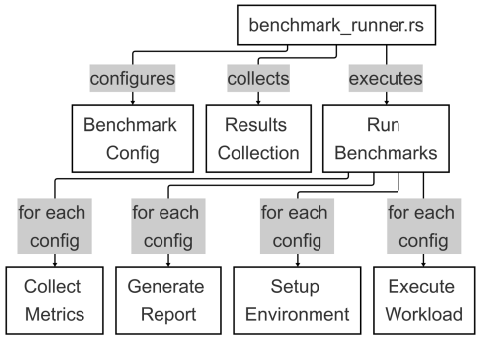

FIGURE 7. Benchmark module - Benchmark Runner.

V. CONCLUSION

This paper has presented Crust, a novel rust framework designed to facilitate the development, validation, and benchmarking of Conflict-free Replicated Data Types (CRDTs). Motivated by the identified gaps in existing CRDT literature and frameworks, and grounded in the rigorous mathematical theory of CRDTs, Crust delivers a comprehensive and extensible platform for creating robust, consistent, and high-performing CRDT.

The primary contribution of this work is the design and thorough exposition of Crust's modular architecture, which includes the Core, Config, Network, Validation, and Benchmark modules. This structure ensures a clear separation of responsibilities, fostering maintainability, scalability, and usability. The Core Module provides the essential CRDT functionalities and synchronization settings, while the Config Module streamlines essential CRDT operations and synchronization settings, while the Config Module simplifies cloud-native deployment and management via Kubernetes integration. The Network Module guarantees dependable communication between replicas. Most importantly, the Validation Module provides an advanced, trait-based framework for verifying the correctness of CRDT implementations against their theoretical principles, filling a critical gap in existing CRDT tools. Alongside this, the Benchmark Module offers a structured and adaptable environment for performance assessment and resource analysis, supporting data-driven enhancements in CRDT-based systems.

By integrating these modules, Crust introduces several key advancements: First, it provides a cloud-native, deployable framework suited for contemporary distributed environments with Kubernetes support. However, it is important to note that effectively leveraging its full deployment capabilities requires developers to possess a basic understanding of Kubernetes concepts and operations. Second, it offers unparalleled validation features, allowing developers to thoroughly confirm the correctness and consistency of CRDT

implementations by embedding mathematical principles into its design and validation processes, thus bridging theory and practice. Third, it also provides a comprehensive benchmarking module that facilitates performance optimization and informed decision-making regarding CRDT selection and synchronization strategies.

Although this paper has outlined Crust's architecture and design in detail, future efforts will focus on critical next steps. First, addressing the security weaknesses in current CRDT frameworks will be a top priority. This will involve exploring security mechanisms and extension traits within Crust. Second, the Benchmark Module for comprehensive empirical studies was utilized, evaluating a wide range of CRDT implementations across diverse synchronization methods, modes, workloads, and network scenarios. These benchmark reports will offer deep insights into performance trade-offs and establish best practices for CRDT deployment. Third, To mitigate the challenges of extending the framework with potentially disruptive features, future work will explore incorporating mechanisms like Rust's feature gates. This would allow for managing complexity and enabling optional functionalities without altering the core implementation principles for all CRDT implementations. Finally, higher-level APIs will be developed and developer tools on top of Crust to further simplify the integration and use of CRDTs in real-world distributed systems. These advancements and architectural considerations will reinforce Crust's role as an essential resource for the CRDT community and developers of resilient distributed systems.

In summary, the Crust framework represents a significant step forward in the development and application of CRDTs. With its sturdy architecture, integrated validation and benchmarking tools, and strong theoretical foundation, it stands as a promising platform for crafting next-generation consistent and efficient CRDTs. It is confidently asserted that Crust provides a significant contribution to the field, being utilized as both a practical tool for developers and a solid foundation for continued research and innovation in CRDT-based systems.

REFERENCES

[1] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds., 2011, pp. 386–400.

[2] Y. Mahéo, F. Guidec, and C. Noûs, "CRDT-based collaborative editing in OppNets: A practical experiment," in *Proc. IARIA*, Sep. 2023, p. 13.

[3] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Yjs: A framework for near real-time P2P shared editing on arbitrary data types," in *Engineering the Web in the Big Data Era* (Lecture Notes in Computer Science), P. Cimiano, F. Frasincar, G.-J. Houben, and D. Schwabe, Eds., Cham, Switzerland: Springer, 2015, pp. 675–678.

[4] L. André, S. Martin, G. Oster, and C.-L. Ignat, "Supporting adaptable granularity of changes for massive-scale collaborative editing," in *Proc. 9th IEEE Int. Conf. Collaborative Comput., Netw., Appl. Worksharing*, Oct. 2013, pp. 50–59.

[5] R. Brown, Z. Lakhani, and P. Place, "Big(ger) sets: Decomposed delta CRDT sets in riak," in *Proc. 2nd Workshop Princ. Pract. Consistency Distrib. Data*, Apr. 2016, pp. 1–5.

[6] N. Saquib, C. Krintz, and R. Wolski, "Log-based CRDT for edge applications," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Sep. 2022, pp. 126–137.

[7] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency in distributed systems," in *Proc. ACM Program. Lang.*, Oct. 2017, pp. 1–28. [Online]. Available: https://github.com/trvedata/crdt-isabelle

[8] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based CRDTs by delta-mutation," in *Networked Systems*, A. Bouajjani and H. Fauconnier, Eds., Cham, Switzerland: Springer, 2015, pp. 62–76.

[9] G. Zakhour, P. Weisenburger, and G. Salvaneschi, "Type-checking CRDT convergence," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, pp. 1365–1388, Jun. 2023.

[10] A. Rinberg, T. Solomon, R. Shlomo, G. Khazma, G. Lushi, I. Keidar, and P. Ta-Shma, "DSON: JSON CRDT using delta-mutations for document stores," *Proc. VLDB Endowment*, vol. 15, no. 5, pp. 1053–1065, Jan. 2022, doi: 10.14778/3510397.3510403.

[11] S. Weiss, P. Urso, and P. Molli, "Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks," in *Proc. 29th IEEE Int. Conf. Distrib. Comput. Syst.*, Jun. 2009, pp. 404–412.

[12] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *J. Parallel Distrib. Comput.*, vol. 71, no. 3, pp. 354–368, Mar. 2011.

[13] W. Yu and C.-L. Ignat, "Conflict-free replicated relations for multi-synchronous database management at edge," in *Proc. IEEE Int. Conf. Smart Data Services (SMDS)*, Oct. 2020, pp. 113–121.

[14] Y. Mao, Z. Liu, and H.-A. Jacobsen, "Reversible conflict-free replicated data types," in *Proc. 23rd ACM/IFIP Int. Middleware Conf.*, Nov. 2022, pp. 295–307, doi: 10.1145/3528535.3565252.

[15] V. B. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency for conflict-free replicated data types," in *Proc. 25th Automated Reasoning Workshop*, 2018, p. 25.

[16] M. Ahmed-Nacer and P. Urso, "A framework for performance evaluation of decentralized eventual consistency algorithms," *EAI Endorsed Trans. Collaborative Comput.*, vol. 3, no. 11, Jun. 2017, Art. no. 152755.

[17] D. J. V. de Sousa, "Security in conflict-free replicated data types," Ph.D. dissertation, 2023.

[18] Rust-Crdt. *GitHub-Rust-Crdt/Rust-Crdt: A Collection of Well-Tested, Serializable CRDTs for Rust*. Accessed: Feb. 20, 2025. [Online]. Available: https://github.com/rust-crdt/rust-crdt

[19] Toeverything. *GitHub-Toeverything/AFFiNE: There Can be More Than Notion and Miro. AFFiNE is a Next-Gen Knowledge Base That Brings Planning, Sorting and Creating All Together. Privacy First, Open-Source, Customizable and Ready to Use.* Accessed: Feb. 10, 2025. [Online]. Available: https://github.com/toeverything/AFFiNE

[20] Pubkey. *GitHub-Pubkey/Rxdb: A Fast, Local First, Reactive Database for JavaScript Applications*. Accessed: Feb. 10, 2025. [Online]. Available: https://rxdb.info/

[21] Dmonad. *GitHub-Dmonad/Crdt-Benchmarks: A Collection of CRDT Benchmarks*. Accessed: Feb. 20, 2025. [Online]. Available: https://github.com/dmonad/crdt-benchmarks

[22] Davidrusu. *GitHub-Davidrusu/BFT-Crdts: Byzantine Fault Tolerant CRDT's and Other Eventually Consistent Algorithms*. Accessed: Feb. 15, 2025. [Online]. Available: https://github.com/davidrusu/bft-crdts

YUNRUI ZHU received the bachelor's degree in computer science from Miami University, OH, USA. He is currently pursuing the master's degree with Auckland University of Technology, Auckland, New Zealand.

JING MA (Member, IEEE) received the Ph.D. degree in computer sciences from Auckland University of Technology, Auckland, New Zealand, in 2019. She is currently a Lecturer with the Department of Computer Sciences and Software Engineering, Auckland University of Technology.