Understanding and Implementing Distributed Glitch-Free Protocol in Meerkat

Introduction

This document presents our comprehensive analysis of the distributed glitch-free protocol for the Meerkat reactive programming language. We begin by explaining our initial implementation approach, then detail our analysis of Julia's distributed protocol, and finally outline the architectural changes required to transition from a centralized to a fully distributed system.

Part 1: Our Initial Implementation - The Centralized Approach

Understanding the Glitch Problem

In reactive programming systems, a glitch occurs when a derived variable temporarily displays an inconsistent intermediate state during an update. Consider a simple example:

```
var a = 1;
var b = 1;
def sum = a + b;  // sum = 2
```

When we execute a transaction that updates both variables:

```
action update { a = 2; b = 2; }
```

Without glitch-freedom guarantees, the following sequence could occur:

1. Variable `a` updates to 2

2. The `sum` reactive observes `a=2, b=1` and computes `sum=3` (incorrect intermediate state)
3. Variable `b` updates to 2
4. The `sum` reactive observes `a=2, b=2` and computes `sum=4` (correct final state)

The value `sum=3` represents a glitch—a temporary inconsistent state that violates the invariant that `sum` should always equal `a + b`. In distributed systems where other components might observe and act upon these values, such glitches can lead to incorrect behavior and difficult-to-debug issues.

Our Centralized Solution

We implemented a glitch-free protocol using a centralized Manager actor that explicitly coordinates all glitch-free derived variables. The approach works as follows:

Language Extension: We added an `@glitchfree` annotation to the Meerkat language, allowing developers to mark specific derived variables that require glitch-freedom guarantees:

```
@glitchfree def y = x + 1;
```

Coordination Protocol: When a transaction modifies variables, the Manager identifies which glitch-free derived variables are affected. Instead of allowing these variables to immediately publish their computed values, they buffer the results and send acknowledgment messages to the Manager. Only after the Manager receives acknowledgments from all affected glitch-free variables does it send explicit commit messages, allowing them to publish their buffered values simultaneously.

Message Flow: The complete flow involves eight distinct message exchanges:

1. Client sends action request to Manager
2. Manager sends write request to VarActor
3. VarActor confirms write completion to Manager
4. Manager releases locks on VarActor
5. VarActor publishes PropChange to DefActor
6. DefActor computes new value, buffers it, and sends GlitchFreeCommitAck to Manager
7. Manager sends GlitchFreeCommit to DefActor
8. DefActor publishes buffered value
9. Manager notifies Client of transaction completion

Implementation Details: We created a `GlitchFreeCoordinator` structure within the Manager that tracks which glitch-free DefActors are involved in each transaction. This coordinator maintains two sets: `expected_acks` (the DefActors we're waiting for) and `received_acks` (those that have responded). When these sets match, the Manager knows all affected variables are ready and sends the commit signal.

Why This Approach is Problematic

While our centralized implementation successfully prevents glitches and passes our test cases, it fundamentally conflicts with Meerkat's architectural goals. The Manager becomes a central coordinator and potential bottleneck. Every glitch-free transaction requires the Manager to track state, wait for acknowledgments, and coordinate commits. This creates several issues:

Scalability: As the number of glitch-free derived variables increases, the Manager must track more state and coordinate more actors, creating a bottleneck that limits system throughput.

Fault Tolerance: The Manager represents a single point of failure. If the Manager crashes during coordination, the entire glitch-free protocol fails.

Architectural Misalignment: Meerkat aims to demonstrate that reactive systems can achieve strong consistency guarantees (glitch-freedom and causal consistency) in a fully distributed manner, without relying on central coordination. Our centralized approach, while functional, does not advance this research goal.

Part 2: Julia's Distributed Protocol - The BasisStamp Mechanism

Core Insight: Dependency Tracking Instead of Coordination

Julia's protocol achieves glitch-freedom through an entirely different mechanism. Rather than having a central coordinator explicitly synchronize updates, the system tracks causal dependencies and allows values to automatically wait for their dependencies to be satisfied. This is accomplished through a data structure called `BasisStamp`.

Understanding BasisStamp

A BasisStamp is a mapping from reactive variable addresses to iteration numbers. It answers the question: "Which versions of which source variables does this value depend on?"

```
pub struct BasisStamp {
    pub roots: HashMap<ReactiveAddress, Iteration>,
}
```

Every value in the system carries a BasisStamp that identifies its causal dependencies. When a variable `x` is written, its iteration number increments. When a derived variable `y = x + 1` computes a new value based on `x` at iteration 5, the resulting value carries a BasisStamp indicating it depends on `x` at iteration 5.

Example with Transitive Dependencies:

Consider this scenario:

```
var x = 0;
def y = x + 1;
def z = y * 2;
```

When `x` is written with value 5:

- `x` gets iteration 1, publishes value 5 with BasisStamp `{x: Iteration(1)}`
- `y` receives this update, computes 6, and publishes with BasisStamp `{x: Iteration(1)}`
- `z` receives `y` 's update, computes 12, and publishes with BasisStamp `{x: Iteration(1)}`

Notice that `z` 's BasisStamp still references `x` , even though `z` doesn't directly depend on `x` . The BasisStamp tracks the transitive closure of dependencies—the "root" variables that ultimately determine this value.

How Glitch-Freedom Emerges

The key to glitch-freedom lies in how derived variables process incoming updates. Instead of immediately computing and publishing new values when any input changes, derived variables buffer incoming updates and only compute when they have a consistent set of inputs.

The Buffering Mechanism: Each derived variable maintains a buffer of incoming updates, where each update includes both the new value and its BasisStamp. When an update arrives, it's added to the buffer rather than immediately applied.

The Consistency Check: Before computing a new value, the derived variable searches its buffered updates for a "consistent batch"—a set of updates where all dependencies are satisfied at compatible versions. This is implemented in the `find_and_apply_batch()` method.

Detailed Algorithm: The algorithm works as follows:

For each buffered update (starting with the oldest):

1. Extract the BasisStamp from this update
2. For each input variable that this derived variable depends on:
    - Check if we have that input at a version that satisfies the BasisStamp requirements
    - If not, look for additional buffered updates from that input that would satisfy the requirement
    - If we can't find sufficient updates, this batch is not yet consistent—try the next seed update
3. If all inputs have consistent versions, we have found a valid batch
4. Apply all updates in this batch, compute the new value, and return it with an accumulated BasisStamp

Concrete Example:

Consider `def z = x + y` where both `x` and `y` are being updated:

```
Initial state: x=1, y=1, z=2

Transaction writes: x=5, y=3

Update sequence:
1. x publishes: value=5, basis={x: Iter(2)}
   - z receives and buffers this update
   - z checks: "Do I have y at the right version?"
   - z's current y is at Iter(1), but the basis doesn't require a specific y version yet
   - z cannot compute yet (missing y update)

2. y publishes: value=3, basis={y: Iter(2)}
   - z receives and buffers this update
   - z now has: x=5 with basis={x:2}, y=3 with basis={y:2}
   - z checks: "Do I have all dependencies?"
   - YES! Both x and y are available
   - z computes: 5 + 3 = 8
   - z publishes: value=8, basis={x:2, y:2}
```

The crucial point is that `z` never computed or published the intermediate value that would result from `x=5, y=1`. The basis checking mechanism automatically prevented the glitch.

Topological Propagation

Julia's protocol also ensures that updates propagate in topological order—variables update before their dependents. The Service actor maintains a topological ordering of all reactive variables and processes them in this order during propagation. This ensures that when a derived variable checks for consistent updates, the updates from its inputs have already been processed and are available.

No Central Coordinator Required

Notice that in this entire process, there is no central coordinator. The Service actor manages its local reactive variables and handles propagation, but it doesn't coordinate between different reactive variables in the way our Manager does. Each reactive variable independently checks its buffered updates and decides when it has a consistent batch. The glitch-freedom property emerges from the local decisions made by each reactive variable based on BasisStamp checking.

Part 3: Architectural Comparison

Message Protocols

Our Centralized Protocol requires explicit coordination messages:

- `GlitchFreeCommitAck` : DefActor → Manager ("I'm ready")
- `GlitchFreeCommit` : Manager → DefActor ("Everyone commit now")

These messages exist solely for coordination purposes and represent overhead that scales with the number of glitch-free variables.

Julia's Distributed Protocol uses only propagation messages:

- `Propagate` : VarActor → DefActor (or Service → Service)

Each propagation message carries a BasisStamp, but there are no additional coordination messages. The protocol overhead is constant regardless of the number of glitch-free variables.

State Management

Our Centralized Approach requires the Manager to maintain:

- A `GlitchFreeCoordinator` for each active transaction involving glitch-free variables

- Knowledge of which DefActors are glitch-free
- The dependency graph to determine which DefActors are affected by each transaction

Julia's Distributed Approach distributes this state:

- Each reactive variable maintains its own buffer of incoming updates
- Each reactive variable knows its own dependencies (inputs)
- The Service maintains a topological ordering, but this is local information

Failure Modes

Centralized Failure: If the Manager crashes while coordinating a glitch-free transaction, the DefActors are left waiting indefinitely for a commit message that will never arrive. Recovery requires detecting the Manager failure and aborting all pending glitch-free transactions.

Distributed Failure: If a Service crashes, only the reactive variables it manages are affected. Other Services continue operating normally. Recovery involves restarting the Service and re-establishing subscriptions, but there's no system-wide coordination failure.

Part 4: Implementation Transition Plan

Phase 1: Introducing BasisStamp Infrastructure

The first step is to add BasisStamp tracking throughout the system without yet changing the coordination mechanism. This allows us to build and test the infrastructure incrementally.

Type Definitions: We define the core types in the message module:

```rust
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct BasisStamp {
    pub roots: HashMap<ReactiveAddress, Iteration>,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct Iteration(pub u64);

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub struct ReactiveAddress {
    pub service: Address,
    pub id: ReactiveId,
}
```

The `ReactiveAddress` uniquely identifies a reactive variable across the entire system, combining the service address with a local reactive ID.

BasisStamp Operations: We implement essential operations on BasisStamp:

```rust
impl BasisStamp {
    pub fn empty() -> BasisStamp {
        BasisStamp {
            roots: HashMap::new(),
        }
    }

    pub fn add(&mut self, address: ReactiveAddress, iteration: Iteration) {
        match self.roots.entry(address) {
            Entry::Vacant(entry) => {
                entry.insert(iteration);
            }
            Entry::Occupied(mut entry) => {
                // Keep the maximum iteration if we see multiple updates
                *entry.get_mut() = (*entry.get()).max(iteration);
            }
        }
    }

    pub fn merge_from(&mut self, other: &BasisStamp) {
        for (address, iteration) in &other.roots {
            self.add(address.clone(), *iteration);
        }
    }
}
```

The `merge_from` operation is crucial—it combines BasisStamps when computing derived values that depend on multiple inputs.

VarActor Modifications: Each VarActor must track its current iteration and include it in propagated values:

```rust
pub struct VarActor {
    pub name: String,
    pub value: Expr,
    pub iteration: Iteration,
    // ... other fields
}
```

When a VarActor writes a new value, it increments its iteration and creates a BasisStamp:

```
fn write_value(&mut self, new_value: Expr, service_addr: &Address) {
    self.value = new_value.clone();
    self.iteration = Iteration(self.iteration.0 + 1);

    let basis = BasisStamp {
        roots: HashMap::from([(
            ReactiveAddress {
                service: service_addr.clone(),
                id: self.id,
            },
            self.iteration,
        )]),
    };

    self.publish(PropChange {
        from_name: self.name.clone(),
        val: new_value,
        basis,
    });
}
```

Message Protocol Update: The `PropChange` message must carry the BasisStamp:

```
pub enum Msg {
    PropChange {
        from_name: String,
        val: Expr,
        basis: BasisStamp,  // Added field
    },
    // ... other messages
}
```

Phase 2: Implementing Basis Checking in DefActor

This is the core of the transition. We replace the Manager coordination with local basis checking in each DefActor.

DefActor State Extensions: The DefActor needs to maintain buffered updates and track the current basis:

```rust
pub struct DefActor {
    pub name: String,
    pub value: Option<Expr>,
    pub expr: Box<Expr>,

    // Buffering for basis checking
    pub input_buffers: HashMap<String, Vec<StampedValue>>,
    pub current_inputs: HashMap<String, StampedValue>,
    pub current_basis: BasisStamp,

    // ... other fields
}

pub struct StampedValue {
    pub value: Expr,
    pub basis: BasisStamp,
}
```

Receiving Updates: When a DefActor receives a `PropChange`, it buffers the update:

```rust
Msg::PropChange { from_name, val, basis } => {
    let stamped_value = StampedValue {
        value: val,
        basis,
    };

    self.input_buffers
        .entry(from_name)
        .or_insert_with(Vec::new)
        .push(stamped_value);

    // Try to find a consistent batch
    self.try_compute_next_value();

    Msg::Unit
}
```

Basis Checking Logic: The `try_compute_next_value` method searches for a consistent set of inputs:

```rust
fn try_compute_next_value(&mut self) {
    // Get the list of input variables this def depends on
    let input_names = self.get_input_names();

    // Try to find a consistent batch
    if let Some(consistent_inputs) = self.find_consistent_batch(&input_names) {
        // Compute new value with these inputs
        let new_value = self.evaluate_with_inputs(&consistent_inputs);

        // Merge all input bases to get the output basis
        let mut output_basis = BasisStamp::empty();
        for stamped_value in consistent_inputs.values() {
            output_basis.merge_from(&stamped_value.basis);
        }

        // Update current state
        self.value = Some(new_value.clone());
        self.current_basis = output_basis.clone();
        self.current_inputs = consistent_inputs;

        // Publish the new value
        self.publish(PropChange {
            from_name: self.name.clone(),
            val: new_value,
            basis: output_basis,
        });
    }
}
```

Finding Consistent Batches: This is the most complex part of the algorithm:

```rust
fn find_consistent_batch(
    &self,
    input_names: &[String],
) -> Option<HashMap<String, StampedValue>> {
    // Start with current inputs as baseline
    let mut candidate_inputs = self.current_inputs.clone();
    let mut candidate_basis = self.current_basis.clone();

    // Try to incorporate buffered updates
    for input_name in input_names {
        let buffer = self.input_buffers.get(input_name)?;

        // Find the first update in the buffer that we can use
        for update in buffer {
            // Check if this update is compatible with our current basis
            if self.is_compatible(&update.basis, &candidate_basis, input_name) {
                candidate_inputs.insert(input_name.clone(), update.clone());
                candidate_basis.merge_from(&update.basis);
                break;
            }
        }
    }

    // Verify we have all required inputs
    for input_name in input_names {
        if !candidate_inputs.contains_key(input_name) {
            return None; // Missing an input
        }
    }

    Some(candidate_inputs)
}

fn is_compatible(
    &self,
    update_basis: &BasisStamp,
    current_basis: &BasisStamp,
    input_name: &str,
) -> bool {
    // An update is compatible if its basis doesn't conflict with our current basis
    // This is a simplified check; the full algorithm is more sophisticated
    for (root_addr, required_iter) in &update_basis.roots {
        if let Some(current_iter) = current_basis.roots.get(root_addr) {
            if required_iter < current_iter {
                return false; // This update is too old
            }
        }
    }
    true
}
```

Phase 3: Removing Manager Coordination

Once basis checking is working in DefActors, we can remove the centralized coordination from the Manager.

Removing Coordinator State: Delete the `GlitchFreeCoordinator` struct and the `glitchfree_coordinators` HashMap from the Manager.

Simplifying Transaction Completion: The Manager's `UsrWriteVarFinish` handler no longer needs to check for glitch-free coordination:

```
// Before (Centralized)
Msg::UsrWriteVarFinish { txn_id, ... } => {
    if self.all_write_finished(&txn_id) {
        self.release_locks(&txn_id).await;

        if self.glitchfree_coordinators.contains_key(&txn_id) {
            // Wait for Acks...
        } else {
            self.send_transaction_committed(&txn_id).await;
        }
    }
}

// After (Distributed)
Msg::UsrWriteVarFinish { txn_id, ... } => {
    if self.all_write_finished(&txn_id) {
        self.release_locks(&txn_id).await;
        self.send_transaction_committed(&txn_id).await;
    }
}
```

Removing Coordination Messages: Delete the handlers for `GlitchFreeCommitAck` and remove the code that sends `GlitchFreeCommit` messages.

Phase 4: Testing and Validation

We need comprehensive tests to ensure the distributed protocol works correctly.

Basic Test: Our existing `test_basic_glitchfree.meerkat` should continue to pass:

```
service basic_glitchfree {
    var x = 0;
    @glitchfree def y = x + 1;
    pub def set_x_5 = action { x = 5; };
}

@test(basic_glitchfree) {
    do set_x_5;
    assert(y == 6);
}
```

Multi-Variable Test: Test with multiple source variables:

```
service multi_var {
    var a = 0;
    var b = 0;
    @glitchfree def sum = a + b;
    pub def update = action { a = 5; b = 3; };
}

@test(multi_var) {
    do update;
    assert(sum == 8);
}
```

This test is critical because it verifies that the DefActor correctly waits for both `a` and `b` to update before computing `sum`.

Chained Dependencies: Test transitive dependencies:

```
service chained {
    var x = 0;
    @glitchfree def y = x + 1;
    @glitchfree def z = y * 2;
    pub def set_x = action { x = 5; };
}

@test(chained) {
    do set_x;
    assert(y == 6);
    assert(z == 12);
}
```

This verifies that BasisStamps correctly track transitive dependencies and that `z` waits for the correct version of `y`.

Conclusion

The transition from centralized to distributed glitch-free protocol represents a fundamental architectural shift. Rather than relying on explicit coordination through a central Manager, the distributed approach uses causal dependency tracking (BasisStamp) to allow each reactive variable to independently determine when it has consistent inputs.

This approach aligns with Meerkat's research goals of demonstrating that strong consistency guarantees can be achieved in fully distributed reactive systems. While our initial centralized implementation successfully demonstrated understanding of the glitch-freedom problem, the distributed protocol provides a scalable, fault-tolerant solution that advances the state of the art in reactive programming systems.

The implementation requires careful attention to the basis checking algorithm, particularly the logic for finding consistent batches of updates. However, the core insight—that glitch-freedom can emerge from local dependency checking rather than global coordination—simplifies the overall system architecture and eliminates the Manager as a bottleneck and single point of failure.