

Deterministic State Distribution Across Ephemeral Execution Nodes in Controller-Driven Real-Time Systems

Timothy M. Gough
IAMMOGO Intelligence Company, Inc.
Fountain Valley, California, USA
Email: tgough@iammogo.com

*A U.S. non-provisional patent application covering this architecture
was filed on Nov. 25, 2025
(Application No. 19/400,020). Priority date: April 2025*

Abstract—Modern real-time systems exhibit non-deterministic behavior and state divergence due to latency and event-driven architectures that lack guarantees of consistent execution across distributed devices. This paper presents a controller-driven model for deterministic state distribution across ephemeral execution nodes, in which a single authoritative source defines and enforces system state. By eliminating client-side interpretation and distributing state rather than events, the model ensures convergence to a consistent execution outcome across all nodes. We further incorporate DAIOS (Deterministic AI Operating System) and DECTL (Deterministic Ethics-Constrained Transition Law) as enforcement layers that validate state transitions prior to execution. The model is evaluated through SYNKTRON, a working implementation, demonstrating immediate state convergence across heterogeneous devices regardless of join time or network variability. These results indicate that controller-driven deterministic state distribution provides a reliable foundation for real-time system coordination without reliance on persistent client logic or cloud-based synchronization.

TABLE OF CONTENTS

1. Introduction to Deterministic State Systems
2. Controller-Driven Deterministic State Model
3. State Distribution Architecture and Execution Semantics
4. Deterministic Enforcement and Admissibility Framework
5. Real-Time System Behavior and SYNKTRON Implementation
6. Comparative Analysis and Application Domains
7. Limitations, Extensions, and Future Directions

1. INTRODUCTION TO DETERMINISTIC STATE SYSTEMS

Modern real-time systems increasingly operate across heterogeneous devices in environments where latency, variability, and asynchronous communication are unavoidable. Traditional architectures rely on event-driven or message-based paradigms, in which discrete events are transmitted and interpreted independently by receiving nodes. These systems are inherently sensitive to local clocks and network jitter, and do not guarantee convergence across all nodes to an identical system state at a given execution point. As a result, real-time environments frequently exhibit non-deterministic behavior, state divergence, and inconsistent system outcomes.

The core limitation of event-driven systems lies in their reliance on local interpretation. Events describe actions, not outcomes, leaving each receiving node responsible for reconstructing system behavior based on its own timing, execution context, and state history. Under variable network conditions, this produces divergent execution paths, particularly as scale and device heterogeneity increase. From a distributed systems perspective, these models inherently prioritize availability over consistency, leading to 'event drift' where state is eventually consistent rather than immediately identical.

Deterministic state systems address this limitation by shifting from event distribution to state distribution. Instead of transmitting instructions to be interpreted, the system distributes a single authoritative state that defines the expected outcome at a given execution point. All nodes are required to conform to this state without deviation, effectively transforming execution nodes into stateless renderers of system state rather than interpreters of events. This removes ambiguity, eliminates local decision-making, and ensures that system behavior is consistent across all participants.

Within this model, authority is centralized. A controller defines the system state, and a coordinating layer ensures its ordered distribution and enforcement. Execution nodes are ephemeral, maintaining no persistent logic or historical dependency beyond the current state. Nodes may join or leave the system at any time, and upon connection, immediately converge to the current authoritative state. This eliminates the need for event replay or synchronization backlogs, as the system operates on the principle that each state update represents a complete and sufficient description of the desired system outcome.

This approach introduces a distinct operational trade-off. In contrast to distributed consensus systems, deterministic state systems prioritize consistency and immediate convergence within bounded network environments, such as local or controlled infrastructure. Partition tolerance is not the primary design goal; rather, the model assumes a constrained network where deterministic execution can be enforced reliably. Under these conditions, the system achieves predictable, synchronized behavior without the complexity of distributed agreement protocols.

As real-time environments continue to expand in scale and complexity, deterministic state systems provide a structured framework for achieving reliable execution across distributed nodes. By eliminating interpretive logic at the edge and enforcing a single source of truth, this model redefines how real-time coordination can be achieved in systems where timing, consistency, and outcome fidelity are critical.

2. CONTROLLER-DRIVEN DETERMINISTIC STATE MODEL

This section defines the formal model underlying deterministic state distribution in controller-driven real-time systems. The model specifies system entities, authority, execution behavior, and constraints required to guarantee consistent state convergence across distributed nodes.

2.1 SYSTEM DEFINITIONS

Controller

The controller is the sole authoritative source of system state. It defines all state transitions and determines the intended system outcome at each execution point. The controller does not execute state; it defines it.

Server

The server functions as a coordination and distribution layer. It maintains the current authoritative state, enforces ordering of

state transitions, and distributes state to all connected execution nodes. The server does not interpret state; it ensures its consistent propagation.

Execution Node

An execution node is a client device responsible for rendering or executing the current system state. Nodes are ephemeral and stateless, maintaining no persistent logic or historical dependency beyond the most recent state received.

State

State is the complete and authoritative representation of the system at a given execution point. Each state is versioned or sequenced (e.g., S_1 , S_2 , S_3), allowing nodes to determine ordering, detect missed updates, and immediately converge to the most recent valid state. A state update is self-sufficient and does not require historical reconstruction.

2.2 CONTROLLER-DRIVEN AUTHORITY

The model enforces a strictly centralized authority structure. All valid state transitions originate from the controller, and no execution node or intermediary component is permitted to modify, reinterpret, or override state once defined. This controller-driven approach eliminates the need for distributed consensus, negotiation, or conflict resolution between nodes. By establishing a single source of truth, the system guarantees that all nodes receive identical state definitions, enabling deterministic execution across the network.

Authority is explicitly defined and enforced, not emergent.

2.3 EPHEMERAL EXECUTION NODES

Execution nodes operate as ephemeral participants within the system. They do not retain historical state, maintain execution logic, or perform local decision-making. Their sole responsibility is to render or execute the current authoritative state as received.

Because nodes are stateless, they can join or leave the system at any time without affecting global system integrity. Upon connection or reconnection, a node immediately converges to the current system state without requiring event replay or synchronization with prior transitions.

This design reduces system complexity, eliminates client-side divergence, and ensures uniform behavior across heterogeneous devices.

2.4 STATE VS EVENT SEMANTICS

Traditional distributed systems rely on event semantics, where messages describe actions to be interpreted and executed by receiving nodes. In such systems, the final state emerges from the cumulative interpretation of events, which may vary based on timing, ordering, and local execution context.

In contrast, the deterministic model operates on state semantics. The system distributes the complete definition of the desired outcome rather than instructions. Execution nodes do not interpret events; they conform to state.

This removes temporal coupling. Event-based systems are dependent on timing and sequence alignment across nodes. The deterministic state model is coupled only to the authoritative state definition, not to the timing of intermediate steps.

Conceptually, event-based systems resemble a sequence of instructions that must be followed in order, while state-based systems define the final outcome directly. Missing intermediate steps in an event sequence leads to divergence; receiving the current state ensures convergence regardless of prior timing.

2.5 SYSTEM ASSUMPTIONS

The deterministic model operates under the following assumptions:

- The system is deployed within a bounded network environment, such as a local or controlled infrastructure.
- Network latency is variable but finite and does not prevent eventual state delivery.
- Execution nodes are capable of rendering or executing the received state within acceptable performance limits.
- A persistent communication channel exists between the server and execution nodes.
- The controller operates as a reliable and continuous source of state definition.

These assumptions define the conditions under which deterministic convergence is achievable.

2.6 SYSTEM CONSTRAINTS

To preserve deterministic behavior, the model enforces the following constraints:

- **Single Authoritative State**
At any given time, only one state is considered valid. New state updates replace prior state without overlap or merging.

- **Ordered State Transitions**
All state updates are strictly ordered via sequence or version identifiers (e.g., $S_i \rightarrow S_{i+1}$). Nodes use this ordering to detect stale or missed updates and converge to the most recent state.
- **Idempotent State Application**
State updates are idempotent. Reapplication of the same state does not alter system behavior, ensuring stability under retransmission or duplication.
- **Stateless Execution**
Execution nodes must not retain historical dependency or introduce local logic. Behavior is determined solely by the current state.
- **Immediate Convergence**
Upon receipt of a newer state, nodes must replace their current state without delay. Nodes do not attempt to reconstruct intermediate transitions.
- **No Event Replay Dependency**
System operation must not rely on replaying prior events to determine current behavior. Each state update is sufficient to define the system outcome.
- **Graceful Degradation Under Constraint**
If a node cannot fully realize a state due to hardware or performance limitations, it must converge to the closest valid representation without introducing divergent logic. The handling of such conditions is addressed at the enforcement and system behavior layers.

These definitions, assumptions, and constraints establish a deterministic model in which system behavior is defined by centrally controlled, versioned state and uniformly executed across ephemeral nodes without reliance on timing alignment or interpretive logic.

3. ARCHITECTURE AND EXECUTION

This section describes how the deterministic model is realized in practice. It integrates system topology with execution semantics, defining how an authoritative state is constructed, ordered, transmitted, and applied across execution nodes in real time.

3.1 SYSTEM TOPOLOGY

The system follows a controller-driven, centralized topology:
Controller \rightarrow Server \rightarrow Execution Nodes

The controller defines state transitions. The server maintains and distributes the authoritative state. Execution nodes receive and render the current state.

This topology eliminates peer-to-peer communication between nodes and removes the need for distributed consensus. All system behavior is derived from a single authoritative path, ensuring consistent state propagation across the network.

3.2 SERVER AS STATE AUTHORITY AND ROUTER

The server operates as both a state authority and a routing layer. It maintains the current authoritative state and is responsible for enforcing ordering and distribution of all state transitions.

As a state authority, the server:

- Stores the latest valid state (S_{\square})
- Validates sequence ordering
- Ensures that only the most recent state is active

As a router, the server:

- Distributes state updates to all connected nodes
- Supports targeted routing (e.g., subsets of nodes by role, zone, or channel)
- Manages connection lifecycles

The server does not interpret or modify state content. Its role is to ensure consistent propagation and enforcement of controller-defined state.

3.3 COMMUNICATION MODEL

The system relies on persistent, bidirectional communication channels between the server and execution nodes, typically implemented via WebSocket connections.

Key properties of the communication model include:

- Persistent connectivity: Nodes maintain an open channel to receive state updates in real time
- Push-based distribution: State updates are transmitted immediately upon definition
- Stateless client communication: Nodes do not request historical data or perform synchronization queries
- Targeted broadcast: State can be directed to all nodes or specific subsets

This model ensures low-latency delivery and continuous alignment with the authoritative state.

3.4 STATE REPRESENTATION AND DISTRIBUTION

State is represented as a structured, self-contained object:

{ type, payload, target, SEQUENCE }:

- type defines the category of state (e.g., visual, media, control)
- payload contains the data required for execution
- target specifies the intended recipients
- sequence defines the ordering of the state

Each state update represents a complete definition of the desired system outcome. State is distributed as a full snapshot of the intended execution context rather than as incremental instructions.

State updates are transmitted as atomic units. Each update is delivered as a complete frame, ensuring that execution nodes never render a partial or intermediate state. A node either applies the full state or does not apply it.

Because each state is self-sufficient, nodes can immediately execute upon receipt without requiring prior state history.

3.5 DETERMINISTIC ORDERING AND REPLACEMENT

Determinism is enforced through strict ordering and replacement semantics.

- Each state is assigned a monotonically increasing sequence identifier (S_{\square})
- Nodes compare incoming sequence values to their current state
- If $S_{\square+1} > S_{\square}$, the new state replaces the current state
- If $S_{\square+1} \leq S_{\square}$, the update is ignored

This ensures that:

- Only the most recent state is active
- Out-of-order or duplicate updates do not affect execution
- State convergence is guaranteed across nodes

Replacement is atomic. Nodes do not merge states or apply partial updates. The current state is fully replaced by the incoming authoritative state.

Intermediate state loss is an accepted trade-off for deterministic convergence. If network conditions cause earlier states (e.g., S_{10}) to arrive after later states (e.g., S_{12}), stale updates are discarded in favor of immediate alignment with the most recent authoritative state.

3.6 TEMPORAL REFERENCE MODEL

The system operates on a controller-defined temporal reference rather than relying on synchronized physical clocks.

Each state corresponds to a logical execution point, represented by its sequence (S_i). Nodes do not depend on local time to determine execution order; instead, they rely on sequence ordering to maintain consistency.

This removes temporal coupling between nodes. Event-driven systems are dependent on synchronized timing and ordered execution of intermediate steps. In contrast, the deterministic state model is coupled only to the authoritative state definition.

This approach mitigates the effects of network jitter and clock drift. Execution is governed by the ordering of states rather than the timing of their arrival.

Optional timing metadata may be included for rendering coordination, but it does not affect state validity, ordering, or admissibility.

3.7 CONSISTENCY GUARANTEES

The architecture provides the following consistency guarantees:

- Strong state convergence: All nodes converge to the same state (S_i) given sufficient message delivery
- Deterministic execution: Identical state produces identical outcomes across nodes
- Idempotent updates: Reapplication of the same state does not alter behavior
- Immediate convergence on join: Newly connected nodes adopt the current state without replay
- No interpretive divergence: Nodes do not introduce local variation

Consistency is defined as convergence to the most recent authoritative state, not preservation of all intermediate transitions. Under variable network conditions, nodes may skip intermediate states but will always converge to the latest valid state.

Consistency is achieved without distributed consensus, relying instead on centralized authority, atomic state distribution, and deterministic replacement semantics.

3.8 SCALABILITY CHARACTERISTICS

The system scales through central authority combined with stateless execution nodes.

Key scalability properties include:

- Stateless clients: Nodes require no historical synchronization, reducing overhead

- Constant state complexity: Each update represents a complete state, independent of system history
- Horizontal node scaling: Additional nodes increase load on distribution but do not affect state logic
- Targeted distribution: State updates can be scoped to subsets of nodes, reducing broadcast load

The primary scalability constraint lies at the server layer, which must manage connections and distribute state in real time. However, because nodes do not perform computation beyond rendering, system complexity remains centralized and predictable.

This architecture is optimized for environments with high node counts and low-latency requirements within bounded network conditions.

4. REAL-TIME SYSTEM BEHAVIOR AND SYNKTRON IMPLEMENTATION

This section defines the mechanisms that ensure only valid state transitions are permitted and that execution remains deterministic under all conditions. The enforcement layer operates at the execution boundary, validating state prior to application and preventing inadmissible transitions from affecting system behavior.

4.1 PRE-EXECUTION VALIDATION REQUIREMENTS

Deterministic execution requires that all state transitions be validated prior to application. Validation is performed without interpretation and consists of the following checks:

- Sequence validation: Ensures correct ordering relative to the current state ($S_i \rightarrow S_{i+1}$)
- Atomic integrity: Confirms the state is complete and not partially delivered
- Schema validation: Verifies that the state structure conforms to the expected format and is free of corruption or malformation
- Rule compliance: Ensures the state adheres to governing constraints

Validation is binary. A state either satisfies all conditions or is rejected prior to execution.

4.2 SEPARATION OF DISTRIBUTION, EXECUTION, AND ENFORCEMENT

The system is composed of three distinct layers:

- **Distribution Layer**
Responsible for transmitting state from controller to execution nodes
- **Enforcement Layer**
Responsible for validating state transitions prior to execution
- **Execution Layer**
Responsible for rendering or applying state at the node level

This separation ensures that propagation, validation, and execution remain independent. Determinism is preserved by preventing interpretation or ambiguity from entering any layer.

4.3 DAIOS AS EXECUTION AUTHORITY

DAIOS (Deterministic AI Operating System) operates at the execution boundary, acting as the kernel-level authority that determines whether a state transition is permitted to execute.

DAIOS functions as an execution gate:

- It evaluates state admissibility prior to execution
- It enforces binary outcomes (allow or block)
- It guarantees that once a state is admitted, execution proceeds deterministically

Once a state passes through DAIOS, it is considered authoritative at the execution layer. No further validation or interpretation occurs beyond this boundary.

4.4 DECTL AS GOVERNING RULE SYSTEM

DECTL (Deterministic Ethics-Constrained Transition Law) defines the rule system governing admissible state transitions.

DECTL specifies:

- Valid state structures
- Permissible transitions between states
- Prohibited or invalid configurations

These rules are applied prior to execution through DAIOS, ensuring that only compliant states are propagated and executed.

DECTL functions as a constraint system, not an interpretive layer. It defines boundaries within which state must operate.

4.5 ADMISSIBILITY OF STATE TRANSITIONS

A state transition is considered admissible if it satisfies all structural, sequential, and rule-based constraints.

Admissibility criteria include:

- Correct sequence ordering
- Atomic completeness of state
- Valid schema conformance
- Compliance with DECTL rules

If a state fails any criterion, it is rejected and not executed.

Rejection results in a no-operation (No-Op) at the execution layer. The node maintains its current valid state (S_{\square}) until a subsequent valid state ($S_{\square+1}$) is received. This prevents undefined behavior, state corruption, or execution failure due to invalid transitions.

No partial execution, fallback interpretation, or error propagation is permitted.

4.6 CLOSED-LOOP EXECUTION MODEL

The system operates as a closed-loop execution model:

Controller \rightarrow DECTL \rightarrow DAIOS \rightarrow Execution Node

- The controller defines state
- DECTL defines admissibility rules
- DAIOS enforces validation at the execution boundary
- Execution nodes apply only validated state

This loop ensures that all state transitions are:

- Defined centrally
- Validated prior to execution
- Executed deterministically

By placing enforcement between definition and execution, the system guarantees that only valid, consistent, and admissible states are realized across all nodes.

5. OPERATIONAL DYNAMICS AND IMPLEMENTATION

This section describes how the deterministic model behaves under real-world conditions and validates its feasibility through a working implementation. It focuses on runtime behavior, node lifecycle, and observed system characteristics, demonstrating how deterministic state distribution performs in practice.

5.1 RUNTIME BEHAVIOR IN REAL-TIME ENVIRONMENTS

At runtime, the system operates as a continuous state propagation loop driven by the controller. Each new state (S_{\square}) replaces the previous state and is distributed immediately to all connected execution nodes.

Execution nodes follow a strict lifecycle upon state receipt:

1. Receive state frame
2. Validate sequence and structure
3. Apply state atomically
4. Render or execute payload

This process occurs without interpretation, buffering, or dependency on prior state history. Execution is triggered directly by receipt of a valid state, ensuring minimal delay between definition and realization.

The stateless nature of execution nodes ensures that runtime behavior remains consistent regardless of prior execution context. As a result, the system does not accumulate execution drift over time.

5.2 LATENCY AND PROPAGATION CHARACTERISTICS

Latency in the system is governed by:

- Network transmission time (controller \rightarrow server \rightarrow node)
- Node processing time (validation + render)

The model does not attempt to eliminate latency but instead isolates its impact. Since execution is based on state rather than event sequences, variability in arrival time does not affect correctness.

Nodes may receive state updates at slightly different times due to network conditions, but all nodes converge to the same final state (S_{\square}). Temporal alignment is approximate, while logical consistency is strict.

Network jitter affects when a state is rendered, not what state is rendered.

5.3 NODE LIFECYCLE

Execution nodes are ephemeral and may join or leave the system at any time without coordination.

- Join: A node establishes a connection to the server
- Leave: A node disconnects without affecting global state
- Reconnection: A node re-establishes connection and resumes participation

The system does not track node-specific state history. Node lifecycle events do not trigger redistribution of prior states or require synchronization protocols.

This design removes lifecycle complexity and prevents cascading effects from node instability.

5.4 STATE RECOVERY AND IMMEDIATE CONVERGENCE

Upon connection or reconnection, a node receives the current authoritative state (S_{\square}) and immediately applies it.

No event replay, backlog processing, or historical reconstruction is performed. Recovery is achieved through direct convergence to the latest state.

This behavior is enabled by:

- Self-contained state representation
- Idempotent state application
- Stateless execution nodes

As a result, nodes can recover instantly regardless of how many prior state transitions were missed.

5.5 SYNKTRON IMPLEMENTATION MAPPING

The proposed model is implemented in SYNKTRON, a working system that demonstrates controller-driven deterministic state distribution in a live environment.

Mapping of model components:

- Controller: Operator interface defining state transitions
- Server: Node.js-based coordination layer managing state and distribution
- Execution Nodes: Browser-based clients rendering state via WebSocket connection

State updates are transmitted as atomic frames containing type, payload, target, and sequence fields. The server maintains the latest state and distributes updates in real time to connected nodes.

SYNKTRON operates entirely within a local network environment, ensuring bounded latency and eliminating cloud dependency.

5.6 OBSERVED SYSTEM BEHAVIOR

Observed behavior in SYNKTRON confirms the properties of the deterministic model:

- All nodes converge to the same state regardless of join time
- Nodes do not diverge under variable latency conditions
- Duplicate or out-of-order updates do not affect execution
- Nodes recover instantly upon reconnection

Intermediate state loss was observed under network saturation, but did not affect final convergence. Nodes consistently aligned with the most recent authoritative state.

No state drift or cumulative error was observed over extended operation.

5.7 PERFORMANCE OBSERVATIONS AND VALIDATION

Performance characteristics observed in implementation include:

- Low-latency propagation within local network conditions
- Stable operation across multiple concurrent nodes
- Predictable execution behavior independent of node count

System load scales primarily with the number of active connections and frequency of state updates. Because execution nodes are stateless and do not perform computation beyond rendering, system complexity remains centralized.

Validation confirms that:

- Deterministic convergence is maintained under real-world conditions
- Stateless node design eliminates synchronization overhead
- State-based execution avoids failure modes common in event-driven systems

These results demonstrate that deterministic state distribution is not only theoretically viable but practically achievable in real-time environments.

6. COMPARATIVE ANALYSIS AND APPLICATION DOMAINS

This section contextualizes deterministic state distribution within the broader landscape of distributed systems, evaluates its trade-offs, and identifies domains where its properties provide clear advantages.

6.1 DETERMINISTIC STATE DISTRIBUTION VS EVENT-BASED SYSTEMS

Event-based systems distribute instructions that must be interpreted and executed by receiving nodes. System behavior emerges from the cumulative processing of events, making correctness dependent on timing, ordering, and local execution context. Under network variability, this leads to state divergence, replay dependencies, and synchronization complexity.

Such systems typically provide eventual consistency, where nodes are expected to converge over time through replay, reconciliation, or synchronization mechanisms. Convergence is not immediate and depends on successful delivery and processing of all intermediate events.

Deterministic state distribution replaces event interpretation with direct state enforcement. Instead of transmitting incremental actions, the system transmits the complete definition of the desired outcome. Execution nodes do not reconstruct behavior; they conform to the authoritative state.

This model provides immediate convergence. Upon receipt of the latest valid state, a node aligns instantly with the system's authoritative condition, regardless of prior state or missed transitions.

This shift produces several fundamental differences:

- Event systems are path-dependent; deterministic state systems are outcome-defined
- Event systems rely on eventual consistency; deterministic systems enforce immediate convergence
- Event systems accumulate error; deterministic systems eliminate drift through absolute state replacement

As a result, deterministic state distribution provides stronger guarantees of consistency and predictability, particularly in environments where timing variability is unavoidable.

6.2 TRADE-OFFS (Centralized Authority vs Distributed Models)

The deterministic model prioritizes centralized authority over distributed consensus. This introduces both advantages and constraints.

Advantages include:

- Elimination of consensus protocols and conflict resolution
- Simplified system behavior with a single source of truth
- Predictable convergence without negotiation between nodes

Constraints include:

- Dependence on the availability and correctness of the controller and server
- Reduced tolerance for network partitioning outside bounded environments
- Limited flexibility for node-level autonomy or local decision-making

In distributed systems terms, the model favors consistency and availability within controlled environments, while explicitly deprioritizing partition tolerance. This trade-off is intentional and aligned with use cases requiring strict determinism.

6.3 APPLICATION DOMAINS

Deterministic state distribution is particularly well-suited for environments that require synchronized behavior across multiple devices with minimal latency and no tolerance for divergence.

Representative domains include:

- Live immersive environments
Systems requiring synchronized visual, audio, or interactive output across large numbers of devices
- Real-time venue systems
Coordinated experiences in hospitality, entertainment, and event-driven spaces
- Distributed device orchestration
Control of heterogeneous devices where consistent state must be maintained without persistent client logic
- Human-interactive systems
Applications where users interact with a shared environment and expect consistent system response

In these domains, the ability to enforce a single authoritative state across all nodes provides a clear advantage over event-driven approaches.

6.4 IMPLICATIONS FOR REAL-TIME SYSTEM DESIGN

The deterministic model introduces a shift in how real-time systems are designed and evaluated.

A key implication is the transition from thick clients to ultra-thin clients. Traditional systems rely on thick clients that maintain local state, interpret events, and execute logic independently. This introduces variability and potential divergence across nodes.

In contrast, deterministic state systems reduce execution nodes to ultra-thin clients:

- No persistent state
- No interpretive logic
- No decision-making capability

Nodes function as deterministic renderers of authoritative state. This reduction in client complexity is what enables consistent execution across heterogeneous devices.

Additional implications include:

- Removal of interpretive logic from execution nodes
Eliminates a primary source of divergence
- Reduction of synchronization complexity
State convergence replaces replay and reconciliation mechanisms
- Emphasis on state definition over event sequencing
System correctness is defined by the current state, not the path taken
- Bounded system design
Systems are optimized for controlled environments rather than global distribution

This model suggests that for a class of real-time systems, particularly those operating within bounded environments, deterministic state distribution offers a more reliable and maintainable alternative to traditional event-driven architectures.

7. LIMITATIONS, EXTENSIONS, AND FUTURE DIRECTIONS

This section summarizes the contributions of the deterministic state distribution model, outlines its limitations, and identifies areas for extension and future research.

7.1 SUMMARY OF CONTRIBUTIONS

This paper introduces a controller-driven model for deterministic state distribution across ephemeral execution nodes in real-time systems. The primary contributions are:

- A state-based execution model that replaces event-driven interpretation with authoritative state enforcement
- A centralized authority architecture that eliminates the need for distributed consensus and reconciliation
- A stateless execution paradigm in which nodes function as deterministic renderers of system state
- A deterministic ordering and replacement mechanism ensuring immediate convergence to the latest valid state
- An enforcement framework incorporating DAIOS and DECTL to validate state transitions prior to execution
- A working implementation (SYNKTRON) demonstrating practical feasibility and real-time operation

Together, these contributions define a structured approach to achieving consistent, predictable behavior in distributed real-time environments.

7.2 LIMITATIONS

The deterministic model introduces constraints that define its appropriate domain of use:

- Dependence on centralized authority
The system relies on the availability and correctness of the controller and server. Failure at this level impacts all execution nodes.
- Limited partition tolerance
The model assumes a bounded network environment. In the presence of network partitions, nodes may become isolated from the authoritative state.
- Hardware variability at execution nodes
Differences in device capability may affect rendering performance, leading to variations in timing even when state is consistent.
- Lack of strict temporal synchronization
While logical consistency is enforced, precise time alignment across nodes is not guaranteed without additional synchronization mechanisms.
- Intermediate state loss under network saturation
The replacement model prioritizes convergence to the latest state, allowing intermediate states to be skipped under load.

These limitations reflect intentional design trade-offs in favor of deterministic execution and simplicity within controlled environments.

7.3 FUTURE WORK

Several areas for extension and refinement are identified:

- Latency normalization and synchronization strategies
Techniques for aligning execution timing across nodes while preserving deterministic state behavior
- Multi-node federation
Extending the model to support coordination across multiple authoritative servers or distributed environments
- Enhanced enforcement capabilities
Expansion of DAIOS and DECTL to support more complex validation rules and dynamic policy adaptation
- Integration with external inputs
Incorporating sensors, audio signals, or other real-time inputs into deterministic state generation
- Performance optimization
Scaling the system to support higher node counts and more complex state payloads while maintaining low latency

These directions extend the model beyond its current implementation and explore its applicability in broader system contexts.

7.4 FINAL REMARKS

Deterministic state distribution represents a shift from event-driven interpretation to outcome-defined execution in real-time systems. By centralizing authority, eliminating client-side logic, and enforcing state consistency at the execution boundary, the model provides a reliable framework for synchronized behavior across distributed nodes.

While not intended to replace all distributed architectures, this approach is well-suited to environments where consistency, predictability, and immediate convergence are critical. The combination of deterministic state definition, enforcement, and execution establishes a foundation for building real-time systems that operate without ambiguity, drift, or reliance on complex synchronization mechanisms.

The results presented demonstrate that deterministic execution across ephemeral nodes is both theoretically sound and practically achievable, offering a viable alternative for a defined class of real-time applications.

8. REFERENCES

- [1] T. M. Gough, “*A Universal Framework for Ethical Machine and Human Decision Systems: Deterministic Ethics-Constrained State Transition Law*,” Zenodo, doi: 10.5281/zenodo.17826046.
- [2] T. M. Gough, “*Beyond Probabilistic AI: A Deterministic Framework for Ethical, Explainable, and Fully Offline Machine Decision Systems*,” Zenodo, doi: 10.5281/zenodo.17766645.
- [3] T. M. Gough, “*The Deterministic Unification Model: Completing AI Theory Through State-Transition Computation*,” Zenodo, doi: 10.5281/zenodo.17786897.
- [4] T. M. Gough, “*The Incompatibility of Probabilistic Inference and Authority: Why AI Systems That Guess Cannot Be Trusted With Decisions*,” Zenodo, doi: 10.5281/zenodo.18013406.
- [5] T. M. Gough, “*From Abandoned Determinism to Computational Law: Why Modern AI Governance Fails at Execution-Time*,” Zenodo, doi: 10.5281/zenodo.18025120.
- [6] T. M. Gough, “*A Deterministic State-Transition Architecture for Client-Side Web Execution*,” Zenodo, doi: 10.5281/zenodo.18521379.