Independent Submission
Internet-Draft
Intended status: Informational
Expires: 8 June 2026

Douglas Dohmeyer
Independent Researcher
5 December 2025

ChainSync: A Synchronization Protocol for Strict Sequential Execution in Linear Distributed Pipelines

draft-dohmeyer-chainsync-02

Abstract

ChainSync is a lightweight application-layer protocol that runs over reliable TCP connections to synchronize a fixed linear chain of distributed processes such that they execute their local tasks in strict sequential order and only after every process in the chain has confirmed it is ready. The protocol has four phases: 1) a forward "readiness" wave, 2) a backward "start" wave, 3) a forward "execution" wave, and 4) a backward exit wave.

The design guarantees strict ordering even when nodes become ready at very different times and requires only point-to-point TCP connections along the chain, thus no central coordinator is needed.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Many distributed workflows (pipeline parallelism in machine-learning training, staged data processing, multi-organization business processes, ordered multi-phase computation, etc.) require that tasks execute in a fixed order across different machines, yet must not begin until every participant is ready.

Standard barriers do not enforce execution order. Token-passing or leader-based schemes introduce complexity and single points of failure.

ChainSync solves this with a simple, fully decentralized four-wave algorithm on a line topology that guarantees:

1. No process starts until the entire chain is ready.

2. Execution order is strictly A -> B -> ... -> N.

3. Clean backward-propagating exit after N finishes.

The protocol requires exactly 4(n-1) messages per synchronization round for an n-node chain (one READY and one START per directed link; and one COMPLETE in each direction).

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Topology and Configuration

The processes form a static logical chain:

(Head) A <-> B <-> C <-> ... <-> N (Tail)

Each process knows:

* The IP address and port of its predecessor (Head has none)

* The IP address and port of its successor (Tail has none)

* Whether it is Head, Tail, or intermediate (inferable from the presence/absence of a predecessor/successor)

Each adjacent pair maintains a single persistent bidirectional TCP connection.

1.3. States

State	Meaning
SYNC	Initial state; waiting for READY from the predecessor (Head starts here but moves to READY when locally ready)
READY	Chain segment to the left is ready; has sent READY to the successor (if not Tail)
WATCH	Has propagated START leftward;

	waiting for COMPLETE from the predecessor (if not Head)
START	Currently executing its local task
COMPLETE	Local task finished; has sent COMPLETE to its successor (if any)

Table 1

1.4. Message Types

Messages are simple ASCII text lines terminated by LF. Recommended format:

<COMMAND>[:<ROUND-ID>]\n

Defined commands:

* READY[:<ROUND-ID>]

* START[:<ROUND-ID>]

* COMPLETE[:<ROUND-ID>]

<ROUND-ID> is optional but RECOMMENDED (e.g., UUID) to support multiple concurrent rounds on the same connection. Implementations running only one round at a time MAY omit it.

1.5. Protocol Operation

1.5.1. Phase 1 -- Readiness Collection (Forward Wave)

* Head (A), when locally ready, moves from SYNC to READY and sends READY to its successor.

* Every other node starts in SYNC. When it receives READY from predecessor *and* becomes locally ready, it moves from SYNC to READY and sends READY to successor.

* When Tail (N) enters READY, Phase 2 begins automatically.

1.5.2. Phase 2 -- Start Trigger Propagation (Backward Wave)

* Tail, upon entering READY, sends START to its predecessor and moves to WATCH.

* An intermediate node, upon receiving START from its successor:

  1. Sends START to its predecessor

  2. Moves to WATCH and waits for COMPLETE from its predecessor

* Head, upon receiving START, has no predecessor and therefore moves directly to START and begins execution.

This phase completes in O(n) messages and guarantees every node knows the entire chain is ready before any node starts.

1.5.3. Phase 3 -- Execution Trigger Propagation (Forward Wave)

* A node in WATCH that receives COMPLETE from its predecessor moves to START and begins execution.

* When a node finishes its task, it moves from START to COMPLETE and sends COMPLETE to its successor (triggers successor to start)

Execution order is therefore strictly A -> B -> C -> ... -> N.

1.5.4. Phase 4 -- Backward Propagating Exit (Backward Wave)

* Tail, upon entering COMPLETE has no successor and therefore immediately sends COMPLETE to its predecessor and MAY terminate.

* An intermediate node in COMPLETE that receives COMPLETE from its successor sends COMPLETE to its predecessor and MAY terminate.

* Head, upon receiving COMPLETE from its successor MAY terminate.

The completion of this phase guarantees the Head node knows all nodes have completed execution.

1.6. Waiting in WATCH State

The RECOMMENDED approach is *push-based*: the node simply blocks on read() from the predecessor’s TCP socket. When the predecessor finishes, it pushes COMPLETE. An alternative approach is to poll the predecessor’s TCP socket.

Both approaches are compliant.

1.7. Example Message Flow (A-B-C-D)

RD: READY
ST: START
CM: COMPLETE

```
A.....B.....C.....D
|-RD->|.....|.....| Phase 1
|.....|-RD->|.....|
|.....|.....|-RD->|
|.....|.....|<-ST-| Phase 2
|.....|<-ST-|.....|
|<-ST-|.....|.....| Phase 3
|.....|.....|.....| A starts immediately
|-CM->|.....|.....| A finishes and B starts
|.....|-CM->|.....| B finishes and C starts
|.....|.....|-CM->| C finishes and D starts
|.....|.....|.....| Phase 4
|.....|.....|<-CM-| D finishes
|.....|<-CM-|.....X D exits
|<-CM-|.....X...... C exits
|.....X............ B exits
X.................. A exits
```

2. IANA Considerations

This memo includes no request to IANA.

3. Security Considerations

Connections SHOULD use TLS 1.3. Production deployments SHOULD use mutual TLS with certificate pinning or pre-shared keys to prevent node impersonation.

4. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

Acknowledgements

The author thanks Grok, an AI system developed by xAI, for assistance in drafting portions of this document based on provided specifications, for editing, for suggestions on the ROUND-ID mechanism for concurrent rounds, and the backward propagation of COMPLETE messages to ensure clean termination.

Author’s Address

Douglas Russell Dohmeyer
Independent Researcher
United States of America
Email: douglas.dohmeyer@protonmail.com