Independent Submission
Internet-Draft
Intended status: Informational
Expires: 24 July 2026

Douglas Dohmeyer
Independent Researcher

20 January 2026

ChainSync: A Synchronization Protocol for Strict Sequential Execution in Linear Distributed Pipelines
draft-dohmeyer-chainsync-03

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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 24 July 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

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).

2. 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.

3. 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.

4. 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

5. 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.

6. Protocol Operation

6.1. 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.

6.2. 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.

6.3. 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.

6.4. 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.

7. 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.

8. 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
```

9. Error Handling

This protocol intentionally does not prescribe error handling. There are many potential applications of the protocol, and some applications have mutually exclusive error handling scenarios. The potential applications may vary on both how faults are detected and how to handle a detected fault.

To make the potential scenarios salient, the following examples are provided:

1. Suppose we have a switch stack of four switches (1, 2, 3, 4) where the switches are configured such that the stack can tolerate the loss of any one switch but not any two switches. Further suppose that ChainSync is implemented so that the switches are restarted in a particular order after applying updates say: 1, 2, 3, then 4. Further suppose that during the upgrade, switch 2 experiences a hardware failure and never comes back online. The preference in this situation is likely that the upgrade halts so that no more switches are lost. In such a situation, the successor switch of 2 (i.e. switch 3) could be configured with a timeout waiting for switch 2 to come back online. Moreover, once the fault is detected, the successor switch could send an SNMP trap notifying a network engineer that its predecessor has not come back online.

2. Suppose we have four nodes where ChainSync is implemented to orchestrate the processing of data. Further suppose the processing times of these data are highly variable and the nodes are not expected to restart during processing. In this case, it might be the preference to implement a heartbeat with a timeout shifting the responsibility of the health check to the implementation. If a node in the chain hangs during processing, its successor should not detect a heartbeat and thus know there is a fault in the chain. Moreover, because nodes are not expected to restart, a failed TCP connection would be sufficient to detect a fault in the chain. Even though a fault is detected, it could be handled in two ways. One, the whole chain could terminate and start all over after correcting the faulty node. Two, the progress in processing could be preserved and only the faulty node needs to be reset for the processing to continue.

3. Suppose we have a multi-organizational business process between four companies C, D, E, and F. Further suppose that this multi-organizational business process by design has no error handling implemented but the preference is to engage people to handle faults. Each organization could be responsible for both detecting its faults and notifying the other organizations in the chain. Perhaps by phone.

Thus we have three examples where both detection is handled differently and the handling of faults is different. It should also be clear that the three examples have both failure scenarios where the error detection from one application would not be appropriate for the other and error handling scenarios appropriate for one but not for the other. The ChainSync protocol neither detects faults nor handles errors because the implementation SHOULD both detect faults specific to its problem domain and handle errors specific to its problem domain.

10. IANA Considerations

This memo includes no request to IANA.

11. 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. This protocol implicitly trusts every node in the chain. Mutual TLS does not defend against compromised nodes. It only prevents untrusted nodes from joining the chain.

12. 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