Distributed Systems: Implementation of a Termination Detection Algorithm

Vitor Gonçalo Costa

June 2026

Abstract

This document is a project report to the course Parallel Programming of Évora's University. It starts with an introduction to distributed systems and details the implementation of *Dijkstra-Scholten* termination algorithm.

1 Introduction

A Distributed System is a collection of independent computing elements (nodes or computers) that appear to the user as a single, coherent system. These nodes communicate and coordinate by exchanging messages over a network. In a single-process computing environment, it is trivial to know when a program has finished. In a Distributed System, however, multiple processes are running asynchronously and communicating through messages. A critical challenge is determining whether all computation across all nodes will eventually halt, or if the system is stuck in an infinite loop (a deadlock for example).

Termination detection is the process of building algorithms that can reliably determine if a set of distributed computations have reached a stable state and are guaranteed to terminate. The difficulty stems from the non-deterministic timing and partial failure model. If one node stops communicating, an external observer cannot tell if there was a crash, finished task or waiting on a condition situation.

There are two main approaches to solving this problem involve deciding where the authority or the coordination point resides. *Centralized* approach provides more control over the distributed system, it is used in microservice orchestration and small fault-tolerant cluster computing. *Decentralized* approach is a more resilient solution, used on technologies like blockchain networks and peer-to-peer file sharing.

2 The *Dijkstra-Scholten* algorithm

This algorithm is ideal for systems where computation starts from a **single source node** (the root). When an **active** node sends a message to an **idle** node, the idle node wakes up and becomes the child of the sender, creating a **dynamic computation tree**. As nodes finish their work and their children finish theirs, they send an **acknowledgment** back up the tree. When the root node finally receives acknowledgments from all its children and becomes idle itself, **termination is detected**. Every node stores the following information:

1. $S_i \in \{\text{ACTIVE}, \text{PASSIVE}\}$: The current working state of the node;
2. $parent_i$: A pointer to the node that first woke node i up (the root node is has null);
3. C_i: The deficit counter. Tracks the number of computation messages node i has sent that have not yet been acknowledged.

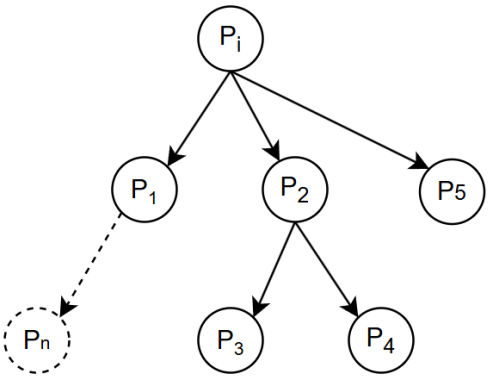

Figure 1: Centralized computation tree of termination detection algorithm.

PASSIVE, waits for message.
ACTIVE, send message to Process n; receives messages from other processes; becomes PASSIVE within a certain probability.
Global termination is detected only at root node, it's made when both of the following conditions are met:

1. $S_i \rightarrow S_r = \{\text{PASSIVE}\}$
2. $C_i \rightarrow C_r = 0$

This is what the function `detect_termination()` does at the core of the algorithm implementation discussed further.

Every process P_n will test the same conditions (global termination) for itself. When a node finises a task sets it's state to passive and check the exit conditions. If true, sends signal to parent and detach from tree by setting the parent to null. When a process receives acknowledgment decrements the deficit count $C_i = C_i - 1$ and re-check the exit condition.

3 The Program

3.1 Implementation details

The `main()` function of a software program spawns two concurrent threads:

1. **basic_threat** → executes a `basic_algorithm()` which sends massages with integer values, switches between active and passive states.
2. **control_thread** → executes the `detect_termination()` function, listening for the *Dijkstra-Scholten algorithm* signals.

When `detect_termination()` returns on process 0 (P_r), the `main()` function assumes the distributed system is finished, kills the `basic_thread`, and exits.

The `basic_thread` triggers hooks on `control-skeleton.c` at the exact same time `control_thread` is reading incoming *Message Passing Interface* [1] control messages. They share local variables, which need to be wrapped with `pthread_mutex_t`

3.2 Behavior analysis

tbd

3.3 Executing the program

tbd

References

[1] Open MPI Development Team. The official open mpi website. `https://www.open-mpi.org/`.