Distributed Systems: Implementation of a Termination Detection Algorithm

Vitor Gonçalo Costa
m70323@alunos.uevora.pt

June 2026

Abstract

This document is a project report to the course Parallel Programming of *Universidade de Évora*. It starts with an introduction to distributed systems and details the implementation of *Dijkstra-Scholten* termination algorithm on an *openMPI* software implementation.

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.

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, a task just finished or still waiting on a condition.

There are two main approaches to solving this problem, they 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* [1] 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:

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

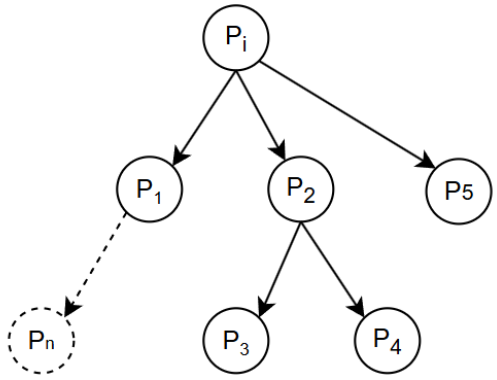

Figure 1: Centralized computation tree of termination detection algorithm.

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 finishes 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

A software program, implementing *openMPI* [2] interface, spawns n MPI processes with the following flow:

1. **Initialization:** The main program initializes MPI with `MPI_THREAD_MULTIPLE` (allowing multiple threads to use MPI safely).
2. **Startup Hook:** The `basic_thread` starts and calls `control_basic_start_hook()`.
3. **Execution Hooks:** As the basic thread loops, it triggers the hooks in `control.c`:
   (a) `control_basic_send_hook()`: It sends a message, so we do C_i++.
   (b) `control_basic_receive_hook()`: It gets a message. If $parent == -1$, it joins the computation tree.
   (c) `control_become_active_hook()`: It starts processing.
   (d) `control_become_passive_hook()`: It runs out of work. It calls `try_resolve_tree()` to see if its deficit is 0.
4. **Background Monitor:** While the hooks are being triggered, the `control_thread` spins in its while loop.
5. **The Final Collapse:** When the root node reaches `PASSIVE_STATE` and $C_i == 0$, its `detect_termination` loop triggers the shutdown sequence

3.1 Implementation details of Termination detection

The `main()` function in `termination.c` spawns two `POSIX` threads per each MPI process:

1. **basic_threat** $\rightarrow$ executes a `basic_algorithm()` that sends messages with some integer value (termination detection algorithms are designed to be payload agnostic) and switches between active and passive states.

2. **control_thread** → executes the `detect_termination()` function, listening for the *Dijkstra-Scholten algorithm* signals.

Because these two threads live in the same process, they share the same memory space. The `basic_thread` updates the variables (via hooks), and the `control_thread` reads them.

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

The `basic_thread` triggers hooks on `control.c` at the exact same time `control_thread` is reading incoming control messages, again, they share local variables, which need to be wrapped with `pthread_mutex_lock(&state_mutex)`, preventing race conditions.

A new MPI tag is defined (`CONTROL_SIGNAL 99`) alongside a custom struct `control_message_t` that holds a type (`MSG_ACK` or `MSG_KILL`). It is completely separate than the one that runs alongside the basic algorithm (`BASIC_MESSAGE 1`), ensuring that the two threads within each process do not intercept each other's network traffic.

- `MSG_ACK` is used when an active node immediately rejects a secondary parent, or when a passive node successfully detaches from the computation tree and signals its parent to decrement its deficit counter (C_i).
- `MSG_KILL` forces all non-root control threads to break out of their infinite listening loops, allowing all processes to reach `MPI_Finalize()` and exit cleanly, essential for preventing deadlocks,

3.1.1 Problems encountered

- **Shared State Problem:** As mentioned previously, to access and read from the shared memory without causing a **race condition** it is required a `pthread_mutex_t`.
- **Premature Termination Bug:** Root node's control thread declared termination before the basic thread boots. The flag $started = true$ was added to prevent this.
- **MPI Deadlock:** Non root nodes were spinning on while loop forever without an acknowledgment signal. The `MSG_KILL` signal was implemented so that the background threads can communicate with each other, it uses `MPI_Iprobe` to non-blocking check for incoming `MSG_ACK` signals from its children.
- ***Passive but in Tree* Bug:** Because a non root node can become idle while $C_i > 0$ it was required to validate $parent == -1$ rather than just checking if the node is passive when joining the tree.

3.2 Behavior analysis

The termination algorithm was subjected to a few executions under different conditions:

- with different amount of spawned processes: With the increase of processes spawned, the algorithm demonstrated that was able to keep up with the basic computations and eventually detected termination.
- with different probabilities on the basic algorithm: By changing the values of `SEND_BASIC_MESSAGE_PROBABILITY` and `BECOME_PASSIVE_PROBABILITY`, the global detection event took different time to be detected, but was eventually reached.

Furthermore, the algorithm's correctness was validated by observing the message counts. In all executions, the total number of basic computation messages sent across the network perfectly matched the total number of control `MSG_ACK` signals received before the root node declared global termination, proving no messages were lost or unacknowledged.

3.3 Executing the program

The code of `Makefile` and `basic.c` was updated in order to take into account a new file `control.c` (previous `control-skeleton.c`). A proper installation of openMPI is needed in order to compile (with *make*) and execute the program. The amount of MPI processes to spawn is provided with flag -np in the command:

```
> mpirun -np 4 ./termination
```

If executing on a personal machine with fewer physical CPU cores than requested processes, the `--oversubscribe` flag is required.

References

[1] Wan Fokkink. Distributed algorithms: An intuitive approach, second edition. Cambridge, Massachusetts; London, England, 2023. MIT Press, The MIT Press.

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