Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
and **Dr. Anna Arpaci-Dusseau** for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

— 1 —

Introduction

1.1 Distributed Shared Memory

To speed up computation on a single machine, we use multiple cores to run OS threads in parallel. Consider the following example:

```
    for (i := 0; i< n; i++) {
        go work(i,results);
    }
```

Here our goal is to run some function `work` on a perhaps large data set of n size. Go easily abstracts this away with the `go` keyword.

Now, what if we want to run this on a distributed system?

```
    package main
    import (
        "net/rpc"
    )
    type Args struct{}
    type WorkServer int64
    func (t *WorkServer) DoWork(args *Args, reply *int64) error {
        // Fill reply pointer to send the data back
        work(args.data, *reply);
        return nil
    }
```

We have to decide now on a system of communication:

- How do we interact with sending and receiving data (conflicts, failures, etc.)
- How should our coordinator dispatch the work?
- What RPCs should we include in our API?

To solve this problem, we reuse a hardware primitive in the previous Section (**??**).

Theorem 1.1: Multithread Comminucation – Virtual Memory

A multithread process can still communicate between threads by utilizing the same virtual memory space, we call this **shared memory**.

The goal is to bring this primitive to the distributed system level.

We clearly define our problem-space:

Definition 1.1: Distributed Shared Memory

A **Distributed Shared Memory (DSM)** system is a system that allows multiple processes on different machines to access a shared memory space as if it were local. I.e,. create the illusion of a single shared memory across multiple machines to enable multithreaded programs in a distrusted setting.

Applications are built on top of a DSM system who abstracts away consensus and communication. DSM systems connect to the underlying hardware support for shared memory (virtual address translation) to coordinate. Page faults are redirected to the DSM system to resolve.

Consensus Model: Strong consistency, and linearizability. **Every** read operation is up-to-date, reflecting the most recent write, if any.

The next solution will be our first draft from which we shall iterate on:

Definition 1.2: DSM – Page Ownership & Swap Protocol (Draft)

Given a system of M_i machines, the Virtual Address Space (VAS) is evenly divided amongst them. I.e., each machine has some partition of pages it owns. On a local machine, the page table will operate as normal; Though, upon a page fault, instead of going to the disk, we redirect to the DSM system. From there we request the missing page over the network.

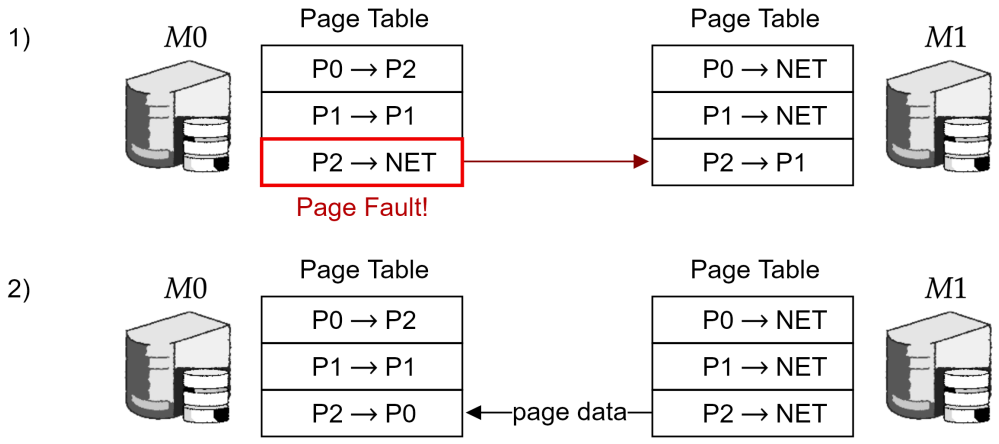

Figure 1.1: 1) System M_0 tries to access P_2 in virtual memory but incurs a page fault, prompting a request to the DSM system. The DSM resolves that M_1 owns the desired page. 2) The DSM performs a network swap, sending the actual page data so M_0 can load it into physical memory.

Though this is a good start, it's expensive:

Theorem 1.2: Sending Pages Over the Network & False Sharing

Pages are often large (4KB), which is expensive to send. If two machines M_0 and M_1 access the same page, but modify different data points (e.g., two different variables), the whole page needs to be sent even though the data is not shared. This is called **false sharing**.

A particular DSM system called **TreadMarks** solves this problem:

Definition 1.3: TreadMarks – Page Ownership & Swap Protocol

Given a system of M_i machines, the VAS is evenly divided amongst them. When page faults occur, the DSM sends **diffs** (change history) over the network instead of the entire page.

This allows for all M_i to start with **RO** (read-only) access on all pages. When M_j wishes to modify a page, M_j notifies all other M_i to invalidate their copies of the page. M_j then gets **RW** (read-write) access to the page.

Each write creates a new V_i version of the page (i, increases monotonically). Where V_i contains the latest changes, and V_{i-1} a snapshot before the changes.

When M_i page faults, the DSM system requests the page from owner M_j, notifying them of the last version M_i saw. The owner M_j locks said page and creates a diff between both M_i and M_j's versions, sending it over the network. Both M_j and M_i revert to RO access.

Consensus Model: Strong consistency, and linearizability.

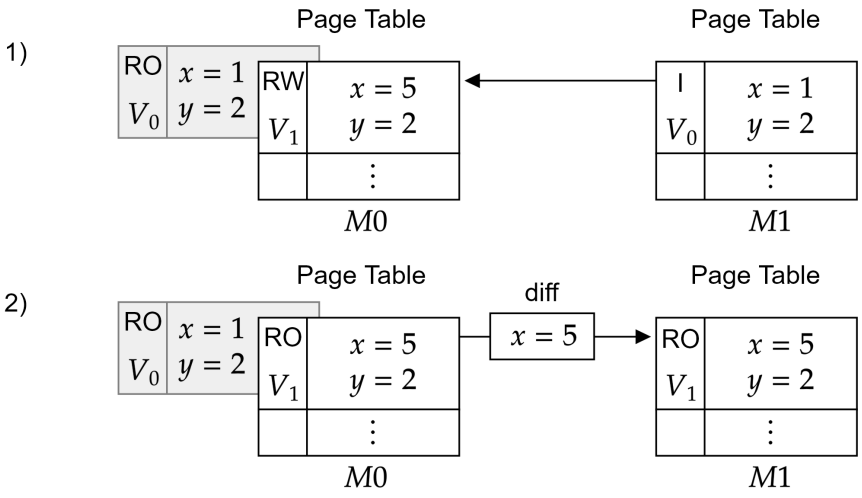

Figure 1.2: 1) $M1$ requests a page from $M2$. 2) $M2$ sends the diff between V_0 and V_1 to $M1$.

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.