Distributed Systems

Diego Oniarti

Anno 2024-2025

Contents

1 Introduction

What is a data center?

Cloud computing pushes the concept further, providing users with generic services, at various layers

The Internet Of Things Internet of things is not only used in domotic systems. A large amount of sensors and actuators are used for industrial use and for "smart cities".

What is a distributes system? We can give a pragmatic definition to: "a collection of independent, autonomous hosts connected through a communication network. Hosts communicate via message passing to achieve some form of cooperation".
This definition gives some good insight, specifying the hosts must be separate, there is message passing instead of shared memory, and so on.

We can also define what is **not** a distributed system, or properties a distributed system does not have:

- shared global clock
- shared memory
- accurate failure detection

Another definition, the 'optimistic' one, describes only how the systems has to appear to an observer: "A collection of independent computers that appears to its users as a single coherent system".

In the same vein, we have a *pessimistic* description: "One in which I cannot get any work done because some machine I have never heard of has crashed".

Why distribution? Distribution of a system can be a requirement as well as a design choice. Some applications are *inherently* distributed, like those that require the propagation of information over a large geographical area, or the collection of data from sensors in different places.

Sometimes it is a design choice, opting to have a system which is fault tolerant.

Defining features Like all other design choices, distributed systems have pros and cons. Some pros are:

- concurrency
- absence of a global clock
- absence of a shared memory

- independent and partial failures

Some cons are:

- higher delays

On the topic of distributed failures, in some distributed systems a failure can be critical, if the failure of a single host can take down the entire system. Other systems with independent hosts, have the opposite behaviour. If an host fails and cannot serve a client, some other host in the network can bare its load.

Challenges of distribution One challenge when implementing a distributed system is **heterogeneity**. The different hosts in a system can have different hardware, programming languages, operating systems, etc.
This problem is tackled through adapters, or most commonly through a layer of middleware.

Another challenge is failure handling. Some key concepts are

- detection: using checksums or what not to detect failures
- masking: hiding from the others that a failure happened in the first place
- tolerance:
- recovery:

In distributed systems we can have **no consensus**. The *two generals problem* is a good example of why we can have no consensus with more hosts.
The two approaches to tackle this problem are:

- change the assumptions: Assume the channels are secure and reliable
- reduce the grantees: Like in TCP

Goals of distribution Two of the main goals when implementing a distributed system are **Openness** and **Security**.
Security is particularly difficult to ensure in a distributed system since it introduces a multitude of new points of failure for an intruder to take advantage of.

Another goal is **scalability**. A scalable system eliminates the performance bottleneck that exists when working on a single host, but are much harder to design.

Finally, **transparency** is the concept of hiding from the exterior how the systems is working. We can hide the allocation of resources, the replication of data, the occurrence of failures, and many other things.
Some things that we can not mask are timezone, delays, etc.
When taken too far, hiding aspects of the system can lead to performance loss.

recap:

1. *ordering* and *causality* enough to ensure working
2. history and partial-history
3. relevant events are *send*, *receive*, and *local* ones

1.1 Happens-Before relation

We say that an evenr e happens before an event e', and write $e \to e'$, if one of the following three cases is true:

1. $\exists p_i \in \Pi : e = e_i^r ...$
2. ...

1.2 logical clocks

A logical clock LC i a function that maps an event e from the history H of a distributed system execution to an element of a time domain T:

$$LC : H \to T$$

Clock consistency

$$e \to e' \implies LC(e) < LC(e')$$

Strong clock consistency

$$e \to e' \iff LC(e) < LC(e')$$

Update rule

$$LC_i = \begin{cases} LC_i + 1 & e_i \text{ is internal or send event} \\ max(LC_i, TS(m)) + 1 & e_i = receive(m) \end{cases}$$

1.3 Causal Historyes and Clocks

The *causal history* of an event e is the set of events that happen-before e, plus e itself.

$$\theta(e) = \{e' \in H | e' \to e\} \cup \{e\}$$

This is theoretically sound but grows too large in a practical setting and is hard to handle.
We would much prefer to always have the *front* of this causal history

1.4 Vector Clocks

The *vector clock* associated to event d is a n-dimenstional vector $VC(e)$ such that

$$VC(e)[i] = c_i$$
$$where \theta_i =$$

Each process p_i maintains a vector VC_i of n elements where every element is the logical lock of a specific process.

Update Rule

$$VC_i[i] = VC_i[i] + 1$$
$$VC_i[j] = max(VC_i[j], TS(m)[j]) + 1 \forall j \neq i$$
$$VC_i[i] =$$

1.4.1 Properties

Less than relation

$$VC < VC' \iff (VC \neq VC') \wedge (\forall k : 1 \leq k \leq n : VC[k] \leq VC'[k])$$

Strong clock condition

$$e \rightarrow e' \iff VC(e) < VC'(e) \iff \theta(e) \subset \theta(e')$$

Concurrent clocks

$$formula$$

Two clocks are concurrent if some elements in a are greater than the equivalent in b and some elements in b are greater than the equivalent in a.

Determining causality

$$formula$$

When I receive a message I increment my value but not that of the other. Sending a message increments my clock before sanding it.

$$\alpha \triangleq \beta$$