Scalable Distributed Computing - Appunti

Francesco Lorenzoni

September 2024

Contents

Part I

Introduction to SDC

Chapter 1

Basic Concepts

1.1 Introduction

Definition 1.1 (Distributed) *Spreading tasks and resources across multiple machines or locations*

Example 1.1.1
- ◇ *Google Search*
- ◇ *Facebook*
- ◇ *Amazon*

Definition 1.2 (Scalable) *Ability to grow and handle increasing workload without compressing performance*

1.2 Motivation for Scalable Systems

- ◇ **Growing data** - large datasets from applications like social media, IoT, AI, etc.
- ◇ **Global Users** - Billions of users worldwide
- ◇ **Performance** - Reducing latency, increasing throughput, and improving reliability
  Sometimes latency is *not* a priority: in some systems it is okay to have high latency to guarantee high throughput and reliability

The challenges are mostly to **manage resources** across geographically distributed systems, and ensuring **low latency** and **high availability**.

1.2.1 Target Architectures

Some architectures which require scalable distributed systems are IoT networks, High-Performance Computing (HPC), and Cloud/Edge Computing.

Example - Cameras in a district

What if I send all the data gathered from cameras to a *single cloud*?

Pros
- ◇ Unlimited storage and processing power
- ◇ Centralized management
- ◇ Simplicity

Cons
- ◇ High latency
- ◇ High bandwidth usage
- ◇ Single point of failure (Scalability and Reliability)

But also simpler applications may considerably benefit from scalable and distributed architectures.
- ◇ Large graph analysis
- ◇ Stream processing
- ◇ Streaming services
- ◇ Machine Learning
- ◇ Big Data
- ◇ Computational Fluid Dynamics
- ◇ Web and online services

1.2.2 Distribution is cool, but...

Consider that local computation is always faster than remote computation. (*Waaay faster*)
From the CPU perspective, time passes *very slowly* when the data travels outside the machine.

If one CPU cycle happened every second, sending a packet in a data center would take 20 hours. Sending it from NY to San Francisco would take 7 years.

1.3 Assessment Method - Exam

There are three options, but note that **in every case an oral exam will follow.**

1. *Writing a Survey or a Report*
2. *Individual or Group Project* (*leq*3 members) Designing, implementing and presenting a solution or prototype related to scalable distributed computing.
3. *Traditional Written Exam* "Questions, answers... you know the drill." Very sad option, in my opinion, but prof. Dazzi did not completely discourage it.

Prof. Dazzi is very open to proposals for the exam, he'd like to stimulate our creativity and curiosity.

Prof. Dazzi says that usually its oral examinations last from 30 to 35 minutes, even though there may be exceptions. Clearly, if the student chooses the report or the project, part of the oral will be about the proposed work, but also questions about the course will be asked.

Chapter 2

Elements and Challenges in Distributed Systems

"A distributed system is one in which the failure of a computer you didn't know existed can render your own computer unusable" — Leslie Lamport

Various definitions of Distributed Systems have been given in the literature, none of them **satisfactory**, and none of them in agreement with any of the others.

Let's try to accept a quite simple one:

" A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system. " — Andrew S. Tanenbaum

From this definition, we can derive two key points:

1. **Autonomy** - Each node in the system is independent and can make decisions on its own
2. **Transparency** - The system appears as a single entity to the user

2.1 Autonomous Computing Elements

Nodes can act independently from each other, so:

- ◇ nodes need to achieve common goals realized by **exchanging messages** with each other
- ◇ nodes **react to messages** leading to further communication through message passing

Having a collection of nodes implies also that each node should know which other nodes are in the system and should contact, and how to reach them. In particular it is important to have a **robust naming system** to identify nodes, and to allow scalability. The key point of a naming system and a DNS is to **decouple** the name of a node from its physical address.

Since each node has its own notion of time, it is not always possible to assume that there is a "global clock": there must be **synchronization** and **coordination** mechanisms to ensure that the system behaves correctly.

2.2 Transparency

The distributed system should appear as a single coherent system, so end users should not even notice that they are dealing with the fact that processes, data, and control are dispersed across a computer network.

This so-called **distribution transparency** is an important design goal of distributed systems.

A —perhaps not-so-perfect— example may be Unix-like operating systems in which resources are accessed through a unifying file-system interface Effectively hiding the differences between files, storage devices, and main memory, but also networks

However, striving for a single coherent system introduces relevant issues: e.g., **partial failures** are inherent to any complex system, in distributed systems they are particularly difficult to hide.

2.2.1 Middleware

To aid the need of easing the development of distributed applications, distributed systems are often organized to have a **separate layer** of software placed on top of the respective operating systems of the computers that are part of the

system.
This layer is called **middleware**. [Bernstein, 1996]

In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network

We list below some example of possible middleware services.

2.2.1.1 RPC - Communication

Remote Procedure Call (RPC) allows an application to invoke a function that is implemented and executed on a remote computer as if it was locally available.

Developers need merely specify the function header and the RPC subsystem generates the necessary code that establishes remote invocations.

Notable examples: Sun RPC, Java RMI, Google RPC (gRPC)

2.2.1.2 Service Composition

Enabling Service composition allows to develop new applications by taking existing programs and gluing them together, e.g. Web services.

An example: Web pages that combine and aggregate data from different sources, e.g. GMaps in which maps are enhanced with extra information from other services.

2.2.1.3 Improving Reliability

Horus toolkit allows to build applications as a group of processes such that any message sent by one process is guaranteed to be received by all or no other process.

As it turns out, such guarantees can greatly simplify developing distributed applications and are typically implemented as part of the middleware.

A more down-to-earth example of improving reliability is the use of RAID systems to ensure that data is not lost in case of disk failure.

Figure 2.1: Horus Toolkit

2.2.1.4 Supporting Transactions on Services

Many applications make use of multiple services that are distributed among several computers.

Middleware generally offers special support for executing such services in an all-or-nothing fashion, commonly referred to as an **atomic transaction**.

Recall all the mess that can happen with threads, race conditions, and deadlocks? Middleware can help with that.

The application developer need only specify the remote services involved.

By following a standardized protocol, the middleware makes sure that every service is invoked, or none at all.

2.3 Building Distributed. . . is it a good idea?

Building distributed systems is a challenging task, and it is not always the best choice. There are **four important goals** that should be met to make building a Distributed System worth the effort:

1. ***Resource*** *accessibility*
2. ***Hide*** *Distribution*
3. *Be* ***open***
4. *Be* ***scalable***

2.3.1 Resource Accessibility

In a distributed system the access, and share, of remote resources is of paramount importance. **Resources** can be virtually anything (peripherals, storage facilities, data, . . .).

Connecting users and resources makes easier to collaborate and exchange information (hint: look at the success of the Internet).

File-sharing used for distributing large amounts of data, software updates, and data synchronization across multiple servers.

2.3.2 Hide Distribution

Hiding distribution is a fundamental goal in the design of distributed systems, and is related to an already mentioned key point, *distribution transparency*, but does not limit to it.
It means to hide that processes and resources are physically distributed across multiple computers possibly separated by large distances.
More precisely, it means to enforce the following properties in **??**:

`ACCESS`	Hide differences in data representation and how an object is accessed
`LOCATION`	Hide where an object is located
`RELOCATION`	Hide that an object may be moved to another location while in use
`MIGRATION`	Hide that an object may move to another location
`REPLICATION`	Hide that an object is replicated
`CONCURRENCY`	Hide that an object may be shared by several independent users
`FAILURE`	Hide the failure and recovery of an object

Table 2.1: Hide Distribution properties

2.3.2.1 Access Transparency

Different systems have different ways to represent and access data. A well designed distributed system needs to hide differences in:

- ◇ physical machine architectures
- ◇ data representation by different operating systems

A distributed system may have computer systems that run different operating systems, each having their own file-naming conventions.

Differences in naming conventions, file operations, or in low-level communication mechanisms with other processes, etc

2.3.2.2 Location and Relocation Transparency

Location transparency means that users can access resources even are not aware where an object is physically located in the system.

Naming plays an important role by assigning logical names to resources, i.e. names not providing information about physical location.

Basically, naming is an indirection process!

Shardcake is a Scala library that provides location transparency for distributed systems, by exploiting "Entity Sharding".

An example of a such a name is the uniform resource locator (URL) http://www.unipi.it, which gives no information about the actual location of University's main Web server.

The entire site may have been moved from one data center to another, yet users should not notice, that is an example of relocation transparency, which is becoming increasingly important in the context of cloud computing.

Actually this is not the case for UniPi hihi ☺.

2.3.2.3 Migration Transparency

Relocation transparency refers just to being moved across the distributed system, migration transparency is:

- ◇ offered by a system when it supports the mobility of processes and resources initiated by users
- ◇ does not affect ongoing communication and operations

A common example is communication between *mobile phones*: regardless whether two people are actually moving, mobile phones will allow them to continue their conversation teleconferencing using devices that are equipped with mobile Internet.

2.3.2.4 Replication Transparency

Resources may be replicated to increase availability or to improve performance by *placing a copy close* to the place where it is accessed.

Hide the existence of copies of a resource or that processes are operating in some form of lockstep mode so that one can take over when another fails.

To hide replication to users, it is necessary that all replicas have the same name.

Systems that support replication transparency should support location transparency as well.

2.3.2.5 Concurrency Transparency

Two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database.

It is important that each user does not notice that the other is making use of the same resource, this phenomenon is called concurrency transparency

- ◇ concurrent access to a shared resource leaves that resource in a **consistent** state
- ◇ consistency achieved through locking mechanisms to give users **exclusive access** to a resource

A more refined mechanism is based on **transactions**, but may strongly impact on scalability.

2.3.2.6 Failure Transparency

Failure transparency is the ability of a system to mask the failures of components from users, so a user —or an application— does not notice that some piece of the system failed and that the system subsequently (and automatically) **recovers** from that failure.

Masking failures is one of the hardest issues in distributed systems and is even impossible when certain assumptions are made.

The main difficulty in masking and transparently recovering from failures is in the inability to distinguish between a dead process and a slowly responding one.

"Is the server is actually down or is the network too badly congested ?" It is often not easy to tell the difference.

2.3.3 Degree of distribution transparency

Distribution transparency is generally considered preferable for any distributed system, however there is a **trade-off** between a *high degree of transparency* and the *performance* of a system.

There are situations in which attempting to blindly hide all distribution aspects from users is *not* a good idea.

examples

- ◇ Internet applications repeatedly try to contact a server before finally giving up, as attempting to mask a transient server failure before trying another one may slow down the system as a whole.
- ◇ Guarantee that replicas, located on different continents, must be consistent all the time, may be costly
  a single update operation may take seconds to complete, that cannot be hidden from users

In other situations it is *not at all obvious* that hiding distribution is "not" a good idea.

- ◇ devices that people carry around and where the very notion of location and context awareness is becoming increasingly important e.g., finding the nearest restaurant
- ◇ when working real-time on shared documents concurrency transparency could hinder the cooperation

There are also other arguments against distribution transparency. Recognizing that full distribution transparency is *impossible*, we should ask ourselves whether it is wise to pretend to achieve it.

In some cases, it may be better to make distribution **explicit** so that: the user and application developer are never tricked into believing that there is such a thing as transparency, resulting in users much better understanding the behavior of a distributed system, and thus prepared to deal with its behavior.

2.4 Openness

An open distributed system is essentially a system that offers components that can easily be used-by, or integrated, into other systems. An open distributed system itself will often consist of components that originate from elsewhere.

Being open enables two key features:

- ◇ Interoperability, composability, and extensibility
- ◇ Separation of policies from mechanisms

Open means that components adhere to standard rules describing the syntax and semantics of those components usually by relying on an Interface Definition Language (IDL). An interface definition allows an arbitrary process that needs a certain interface, to interact with another process that provides that interface. This allows two independent parties to build completely different implementations of those interfaces.

Proper specifications are **complete** and **neutral**. *Complete* means that everything that is necessary to make an implementation has indeed been specified. However... many interface definitions are not at all complete so that it is necessary for a developer to add implementation-specific details. This is just as important is the fact that specifications **do not prescribe what an implementation should look like**, they should be *neutral*.

As pointed out in Blair and Stefani, completeness and neutrality are important for **interoperability** and **portability**.

- ◇ **Interoperability** - characterizes the extent by which two implementations of systems from different manufacturers can work together
  - by relying on each other's services
  - as specified by a common standard
- ◇ **Portability** - characterizes to what extent an application developed for a given distributed system can executed
  - without modification
  - on a different distributed system implementing the same interfaces

Another important goal for an open distributed system is that it should be easy to **configure** the system out of different components (possibly from different developers). Also, it should be easy to **add** new components or replace existing ones without affecting those components that stay in place.
In other words, an open distributed system should also be **extensible**.

For example, in an extensible system, it should be relatively easy to add parts that run on a different operating system, or even to replace an entire file system

2.4.1 Policy and Mechanism Separation

To achieve flexibility in open distributed systems it is crucial that the system be organized as a collection of relatively small and easily replaceable or adaptable components.

This implies to provide definitions not only for the highest-level interfaces, i.e., those seen by users and applications, but also for interfaces to internal parts of the system and describe how those parts interact.

This approach is an alternative to the classical monolithic approach in which components are implemented as one, huge program makes hard to replace or adapt a component without affecting the entire system.

2.5 Scalability

We were used to having relatively powerful desktop computers for office applications and storage.

We are now witnessing that such applications and services are being placed "in the cloud",leading in turn to an increase of much smaller networked devices such as tablet computers.

With this in mind, scalability has become one of the most important design goals for developers of distributed systems.

2.5.1 Dimensions of Scalability

Scalability is a complex issue and can be seen from different perspectives, such as:

- ◇ **Size** - A system can be scalable with respect to its size i.e., we can add more users and resources to the system without any noticeable loss of performance.
- ◇ **Geographical** - A geographically scalable system is one in which the users and resources may be distant, but communication delays are hardly noticed.
- ◇ **Administrative** - An administratively scalable system is one that can still be easily managed even if it spans many independent administrative organisations

2.5.2 Size scalability

Many **users** need to be supported → limitations of centralized services.

Many services are **centralized** → implemented by a single server running on a specific machine or in a group of collaborating servers co-located on a cluster in the same location.

The problem with this scheme is obvious: the server, or group of servers, can become a **bottleneck** due to three root causes:

- ◇ The computational capacity, limited by the CPUs [CPU bound]
- ◇ The storage capacity, including the I/O transfer rate [I/O bound]
- ◇ The network between the user and the centralized service [Network bound]

2.5.3 Geographical scalability

This kind of scalability relates on the difficulties in scaling existing distributed systems that are **designed for local-area networks**, many of them even based on synchronous communication e.g., a party requesting service blocks until a reply is sent back from the server implementing the service.

2.5.4 Administrative scalability