LTAT.06.007 Distributed Systems
Lecture 1 - Introduction to distributed systems

Huber Flores, PhD
ASSOCIATE PROFESSOR

Tartu, Estonia, Delta building (auditorium 1021) 07/02/2022
Agenda

• **Goal:** To define general principles about distributed systems

• **Content:**
  - A distributed systems is… (concept)?
    - Why is necessary a distributed system?
    - When to **avoid** a distributed system?
  - Computer networking
  - System architecture
  - Remote procedure calls (RPC)

**After this lecture, you should be able to:**

• Understand the characteristics of a distributed systems, its components and design principles

• Highlight key technologies to build a distributed system.
Single computer applications

Everything runs in a single computer! Easy and effortless

Keep it simple!
Distributed applications are everywhere!

- Distribution is critical (even in single computers)
- Managing distribution is difficult

1. Multiple cores
2. Multiple disks (storage)
3. Multiple processes (multi-thread)
4. ...

1. Multiple resources
2. Multiple services
3. Multiple types of communication (broadcast, multicast)
4. ...

1. Multiple infrastructures
2. Multiple organizations
3. Multiple technologies
4. ....
A distributed system is…

**Definition**

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

**Other definitions**

- “… a system in which the failure of a computer you didn’t even know existed can render your own computer unusable” - Leslie Lamport

- A network of nodes with independent behavior working towards achieving a common goal

**A few examples of distributed systems are:**

- Google – *for big data processing*;
- Facebook – *for content sharing*;
- Sensor Networks – *for monitoring of artifacts*;
- Surveillance systems – *for security*
A distributed system is

Nodes
Components
Resources
Objects
Processes

Same thing, different context

Independent behavior
Each node is autonomous and will thus have its own notion of time: there is no global clock. Leads to fundamental synchronization and coordination problems.
Why is necessary a distributed system?

The short answer: Performance and availability (achieved through replication)
Why is necessary a distributed system?

Resource sharing
Resource sharing
Why is necessary a distributed system?

• Resource sharing
• Better reliability (availability is not the same as reliability)
• Transparency -> recover from partial failures (unnoticeable)
• Physical security
  ▪ Having a copy of a service in a rack rather than in two different locations
Availability and transparency
Availability and transparency

Internet

Location 1 (Tallinn)

Location 2 (Helsinki)
Why is necessary a distributed system?

- Resource sharing
- Better reliability (availability is not the same as reliability)
- Transparency -> recover from partial failures (unnoticeable)
- Physical security
  - Having a copy of a service in a rack rather than in two different locations
- Solving complex problems (openness)
  - Single computers have processing limits
  - Multiple computer working together

Large Hadron Collider at CERN is supported by a worldwide computing infrastructure with 1 million CPU cores for data analysis (https://wlcg-public.web.cern.ch)
Openness

Achieved through middleware

No communications
Openness

Achieved through middleware (man in the middle)

Translates (Integrates)
When to avoid a distributed system?

- Always! Keep it simple
  - Developing applications that run in a single computer are easy
- Building a distributed system is a hard challenge
  - Communication may fail
  - Nodes (processes) may crash, e.g., software bug, old version
  - Behavior of components is non-deterministic, e.g., RAM or disk gets damaged
Computer networking

**System abstraction**

Node $i$ \[\text{Message } m\] Node $j$

**Simple but complex in practice**

- **Hardware fragmentation:** integration and deployment (fiber, wireless, cellular, devices…)

- **Different networks:** eduroam, Coffee shop, cellular operator, etc.

  The physical method to deliver a message is not important -&gt; network protocols
Computer networking

System abstraction

Simple but complex in practice

- **Protocol**: The foundation of human connection – a shared convention of procedures and expectations

- **Network protocol**: it decides what messages to send, and how to process the messages when they are received.
Communication between two points

- **Circuit switching**: It uses telephone lines, a channel is open between the caller (Node i) and callee (Node j), the channel supplies constant (bi-directional) bandwidth between the parties as long as the call lasts.

- **Packet switching**: Sender (Node i) and receiver (Node j) atomizes messages into tiny shards (“packets”) and combine them into communal flow of data.
Latency and bandwidth

**Latency:** The time that it takes for a message to arrive
- RTT (Round trip time) is a common metric to measure it. For instance, [https://testmy.net/latency](https://testmy.net/latency)
- Same network (same location): 1ms
- Different location: >100 ms

**Bandwidth:** Data volume (to transmit) in a time unit.
- 2G ≈ 64 kbps
- 3G ≈ 2 Mbps
- 4G (LTE) ≈ 50 Mbps
- 5G ≈ 10 Gbps
- 6G? (Hint: AI)

(in practice vary significantly!)
System architecture

Client-server architecture

The Web (distributed system that you use everyday) -> Two types of nodes, server (host websites), client (Web browsers to display the websites)

Node $i$  Client  Server  Node $j$

GET <web page>

<HTML> My website</HTML>

Message $m$

Time flows from top to bottom
System architecture
System architecture

Client-server architecture

Client + Communal (packet) data flow (No dedicated path) + Server
Remote Procedure Calls (RPC)

Example: Online payments

Client -> Online book shop; Server -> Service payment

Online book shop

Charge 15 euros for a book

Success

Service payment
### Remote Procedure Calls (RPC)

**Example: Online payments**

Request (Client) – Response (Server) paradigm

![Diagram showing the process]

- Online book shop
- Service payment
- Transparent process
Remote Procedure Calls (RPC)

Example: Online payments
Request (Client) – Response (Server) paradigm

- Developer (APIs)
- End-user (GUI)

Online book shop
Service payment
Transparent process
Back-end (interfaces)
Example: RPC

```java
1  # Online book payment (client side)
2
3  Order order = new Order();
4  order.setCardNumber("1234 6574 8988 9283");
5  order.setCardExpirationData("08/2025");
6  order.setCardExpirationData("987");
7
8  OnlinePayment result = servicePayment.ProcessPayment(order, 15, Currency.EUROS);
9
10  if (result.isSuccess()){
11      completeOrder();
12  }
```

What functionality is located in different node?
Example: RPC

```java
# Online book payment (Client side)

Order order = new Order();
order.setCardNumber("1234 6574 8988 9283");
order.setCardExpirationData("08/2025");
order.setCardExpirationData("987");

OnlinePayment result = servicePayment.ProcessPayment(order, 15, Currency.EUROS);

if (result.isSuccess()){
    completeOrder();
}
```

What functionality is located in different node?

- Included into client applications through APIs
- Executed as local code
Dissecting RPC

Function starts

Online book shop

processPayment() -> stub

RPC client

Marshalling parameters

m1

Un-marshalling parameters

RPC server

ChargeOrderPayment()

Service payment

Waiting time

Synchronize result to app state

Un-marshalling result

m2

Marshalling result

Function returns

Message m = m1 + m2

(Un) / Marshalling

• What the message contain?
Dissecting RPC

Dissecting RPC

m1 = 

```json
{
  "request": processPayment,
  "order": {
    "number": "123465748989283",
    "expDate": "08/2025",
    "CVC": "987",
  },
  "amount": 15.00,
  "currency": "euros"
}
```

m2 = 

```json
{
  "result": success,
  "transaction_id": "333HBK"
}
```

Args (parameters) matching a format specification

- XML
- JSON (like in the example)
More about RPC

Essence
- Widely-used technique underlying many DS

History
- CORBA (1990s)
- Microsoft’s DCOM and Java RMI (Remote Method Invocation)
- Thrift (Facebook, 2007)
- gRPC (Google, 2015)
- REST (along with JSON) - The most common form of RPC is implemented using JSON data sent over HTTP.
- Ajax (Web browser performance)
RPC in enterprise systems

Essence

• Service-oriented architectures (SOA) -> REST and HTTP
• “Micro services”, splitting large software into multiple small services

Containerization
Tools, e.g., Docker, Kubernetes
RPC in enterprise systems

Essence

- Service-oriented architectures (SOA) -> REST and HTTP
- “Micro services”, splitting large software into multiple small services

How?
By default -> REST
REST APIs

Essence

- HTTP verbs (GET, POST, PUT, DELETE, etc.)
  - Standardized format, CRUD operations
- REST semantics are well understood

Resources are typically named with **nouns**

Resources

Collection [http://foobar.com/api/v1/users](http://foobar.com/api/v1/users)

Entity [http://foobar.com/api/v1/users/user-1](http://foobar.com/api/v1/users/user-1)

Relation [http://foobar.com/api/v1/users/user-1/orders](http://foobar.com/api/v1/users/user-1/orders)
REST APIs

Essence

- Non-crd operations?
  - Archive? -> action (it does not fit into defined verbs)

- Way around?
  - Flags, new sub-operations

```
PATCH ../users/users-1
{
  "archived": true
}
```

```
PUT ../users/users-1/deactivate
```

Archived operation
Deactivate operation
REST APIs

Essence

- Non-crud operations?
  - Archive? -> action (it does not fit into defined verbs)

- Way around?
  - Flags, new sub-operations

PATCH .../users/users-1

```
{
    "archived": true
}
```

PUT .../users/users-1/deactivate

- Big payloads
- Over complexity for easy operations
RPC APIs

Essence

• RPC is about actions (REST is about resources)
  ▪ End-points for actions

https://slack.com/api/chat.postMessage

https://slack.com/api/chat.scheduleMessage

https://slack.com/api/chat.deleteScheduleMessage
RPC APIs

Essence

• RPC is about actions (REST is about resources)
  ▪ End-points for actions

https://slack.com/api/chat.postMessage
https://slack.com/api/chat.scheduleMessage
https://slack.com/api/chat.deleteScheduleMessage

• Difficult to discover
• Limited standardization
• Lead to function explosion
RPC APIs

gRPC example

“Micro services”, splitting large software into multiple small services

How?
By default -> REST
gRPC: in a nutshell

How does it work?

- Service definition to describe the API (aka IDL – Interface Definition Language)
- Client and server generate code
- HTTP 2 support (bi-directional channel, reusable, serializable binaries)
HTTP 2.0

It is all about performance

RPC and threads

Multiple request-response at once

Concurrency managed through **threads**

- **Online book shop**
- **Service payment**

- Charge 15 euros for a book
- Pay 50 euros for a subscription
- Success book
- Success subscription

**Multiple ways of implementing this in practice**
Summary

• Defined what is a distributed system, when to design one, and when to avoid it
• Looked at the fundamental of computer networking and its role for enabling system architectures
• Studied basic communication principles (RPC and its evolution)
• Defined the request/response paradigm and the role of APIs
• Initial thoughts about role of threads in distributed systems
References

Part of this material is inspired by:

- Distributed Systems course given by Dr. Martin Kleppmann (University of Cambridge, UK)
- Introduction to gRPC: A general RPC framework that puts mobile and HTTP/2 first (M. Atamel, R. Tsang)
Next lecture
RPC and threads
Questions?

E-mail: huber.flores@ut.ee