Information Processing

Week 4

Cloud Services, Client-Server, NoSQL Database

Today's Topics

- This presentation describes the activities covered in Lab 4
  - **AWS Connectivity, Client-Server, NoSQL Database**
- The presentation has two parts:
  - **Part 1**: Describes how to make some basic use of a cloud platform to run your server-side apps
    - How to remotely configure a virtual server on the cloud platform (Amazon's AWS is taken as an example)
    - How to make local apps communicate with remote apps running on the cloud
  - **Part 2**: Describes how to use a database on a reliable remote server to implement persistence features in your apps
    - How to configure your compute cloud service to use a distributed NoSQL database on the cloud
    - How to perform basic CRUD (Create, Read, Update, Delete) and other operations on the database from your server-side code

Architecture of a Typical IoT System

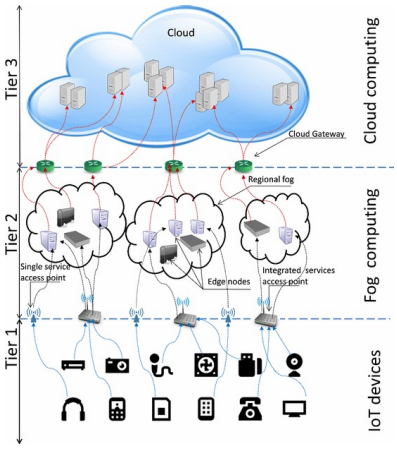
Roughly mapping your current system nodes to the three tiers:

- **Tier 1 — IoT / Edge devices**
  - Closest to the physical system; perform low-latency, hardware-coupled computation and control, e.g., Zynq board (PS + PL with hardware acceleration, etc.)
- **Tier 2 — Fog / Edge Computing**
  - Local intermediary layer for aggregation, preprocessing, control, and cloud communication, e.g., local laptop / local computing component (interface to Zynq PS, AWS client)
- **Tier 3 — Cloud computing**
  - Centralised, elastic compute and storage for coordination, services, and long-term data, e.g., AWS server, AWS database

The centralized Tier-3 cloud layer enables wide and remote access to the application by providing globally reachable services and shared state.

Database belongs in Tier 3 to provide global, persistent, shared state, especially when multiple fog nodes are involved.

Communication between tiers is handled over **networked interfaces**, with Tier-2 acting as the gateway between edge devices and cloud services.



Three-tier architecture of a typical IoT system

Image Source:

Efficient and Dynamic Scaling of Fog Nodes for IoT Devices. Available from: https://www.researchgate.net/figure/Three-tier-architecture-of-a-typical-IoT-system_fig3_317380217 [accessed 12 Jan 2026]

Cloud Computing – What is it?

- "Cloud computing is an information technology (IT) paradigm that enables ubiquitous access to shared pools of configurable system resources and higher-level services that can be rapidly provisioned with minimal management effort, often over the Internet. Cloud computing relies on sharing of resources to achieve coherence and economies of scale, similar to a public utility."
- "Simply put, cloud computing is the delivery of computing services – servers, storage, databases, networking, software, analytics and more – over the Internet ("the cloud"). Companies offering these computing services are called cloud providers and typically charge for cloud computing services based on usage, similar to how you're billed for gas or electricity at home."

https://azure.microsoft.com/en-gb/overview/what-is-cloud-computing/

Source: https://en.wikipedia.org/wiki/Cloud_computing

Cloud Delivery Models

Our Use Case

IaaS: Infrastructure as a Service

The consumer is able to deploy and run arbitrary software, which can include operating systems and applications.

The consumer does not manage or control the underlying cloud infrastructure (hardware) but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

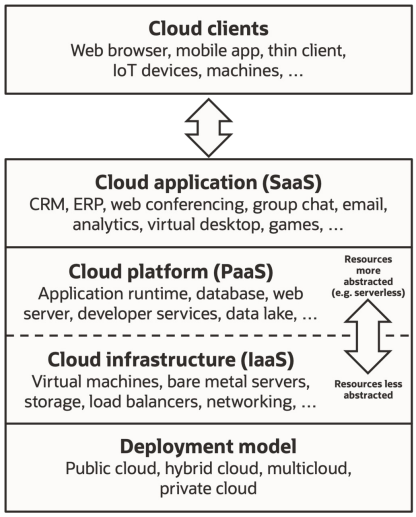


Image source:
https://en.wikipedia.org/wiki/Cloud_computing

SaaS
The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

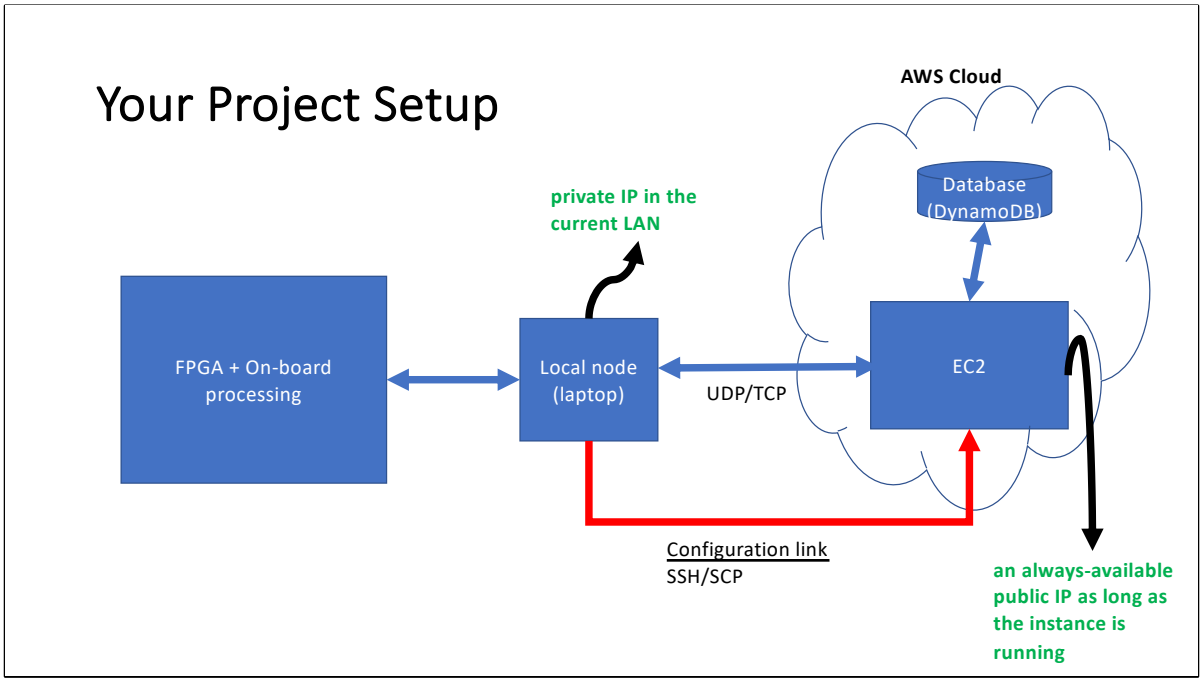
PaaS
The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment. (API provided

by provider).

IaaS
the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure (hardware) but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

We will be using AWS in the IaaS mode.

Your Project Setup



Amazon's Elastic Compute Cloud (EC2)

- **Amazon Elastic Compute Cloud (EC2)** is a part of Amazon.com's cloud-computing platform: Amazon Web Services (**AWS**)
  - lends virtual computers on the cloud to configure and run your apps
  - the virtual computer is typically running a Linux distribution (e.g., Ubuntu)
- Using EC2 an example of IaaS
  - We can choose an OS, storage, computing resources, networking configurations, etc.
- Unlike the lower tiers (Edge and Fog), we can rely on EC2 to be **elastic, i.e.** to grow according to our application's needs

Amazon Web Services, Inc. (AWS) is a subsidiary of Amazon that provides on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered, pay-as-you-go basis.

EC2 Configuration Steps (Lab 4)

Using the web-based console on your AWS account

- Choose **a machine image**: processor, storage, operating system, etc.
- Instance type (**computational power**): no of processors, amount of memory, networking capacity, etc. (Default options on the free-tier suffice)
- Configure **security group**
  - What type of connections are permitted? TCP, UDP, SSH only?
  - What port range is permissible?
  - Are only specific IP addresses allowed to connect, or anyone can connect?
- EC2 is one of the AWS services
  - To communicate with other services, it must do so through a predefined role (IAM role)
  - an **IAM role** is an identity you create in your account that provides specific permissions.
    - E.g. the *DynamoDBAccessforEC2* role in your lab gives your EC2 instance to a DynamoDB database.
- The EC2 instance is **always-running, with a public IP**, unless you explicitly shut it down

Basic Software Installations on EC2 (Lab 4)

You can install any software tools on your EC2 instance that are required by your app to run on Linux:

Some basic tools needed for Lab 4:

- Python
- Pip
- Python packages using pip
- **Boto3**: AWS's Python API to interface with services like DynamoDB, S3 etc.
- g++, etc.

Example: A simple UDP Client, and Server on EC2

```
from socket import *
serverIP = "16.171.58.69" # The public IP of your EC2 instance
serverPort = 12000
clientSocket = socket (AF_INET , SOCK_DGRAM)
message = input ( " Enter a string : " )
clientSocket.sendto (message.encode () , ( serverIP , serverPort ) )
reply , _ = clientSocket.recvfrom (2048) # ignoring sender IP / Port
print ("Server response : " , reply.decode ())
clientSocket . close ()
```

UDP Client on a Local Node

```
from socket import *
serverPort = 12000
serverSocket = socket ( AF_INET , SOCK_DGRAM )
serverSocket.bind(('0.0.0.0', serverPort))
print ( " UDP server running and listening on port " , serverPort )
while True :
    message , clientAddress = serverSocket.recvfrom(2048)
    text = message . decode ()
    if text . isupper () :
        reply = " ALL CAPS "
    else :
        reply = " NOT ALL CAPS "
    serverSocket.sendto(reply.encode(),clientAddress )
```

UDP Server on EC2

Example: A simple TCP Client, and Server on EC2

```
from socket import *
serverIP = "16.171.58.69"   # Public IP of EC2 instance
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverIP, serverPort))
message = input("Enter a string: ")
clientSocket.send(message.encode())
reply = clientSocket.recv(2048)
print("Server response:", reply.decode())
clientSocket.close()
```

TCP Client on a Local Node

Note:
You can either build connectivity in a ground-up fashion using these basic starting points, or
Use a higher-level protocols like HTTP or WebSockets, or more elaborate frame works.

```
from socket import *
serverPort = 12000
welcomeSocket = socket(AF_INET, SOCK_STREAM)
welcomeSocket.bind(('0.0.0.0', serverPort))
welcomeSocket.listen(1)
print("TCP server running and listening on port", serverPort)
while True:
    connectionSocket, clientAddress = welcomeSocket.accept()
    message = connectionSocket.recv(2048)
    text = message.decode()
    if text.isupper():
        reply = "ALL CAPS"
    else:
        reply = "NOT ALL CAPS"
    connectionSocket.send(reply.encode())
    connectionSocket.close()
```

TCP Server on EC2

All your servers may run as daemons on your EC2 instance.

Database

- A database is an organized collection of data
  - Structured so it can be efficiently searched, updated, and managed
- A database is usually accessed through a Database Management System (DBMS)
  - e.g., SQLite, MySQL, PostgreSQL, DynamoDB (a DBMS provides a controlled interface between programs and stored data
- Why not just files?
  - Raw files (e.g. text, CSV, binary) require programs to manually handle reading, writing, parsing, and consistency.
  - Databases store data in structured tables or collections and provide query-based access, handling storage, consistency, and access automatically.

Database Options

1. SQL databases (covered next week)

- Well defined **schema**
  - the database is modeled as a set of joinable tables with pre-defined columns
- Entities are stored in tables, **managed by an RDBMS**
- **Tables can be joined** and using a powerful and flexible querying language: **SQL**
- Data follows are **rigid structure** and is mostly non-redundant
- Changing/updating the structure of data involves change in schema and **costly database alterations**.
- There is a DBA – a database administrator -- between the application developer and the database, responsible for defining and altering the schema if needed.
  - **Using SQL's DDL**

- **Naturally ideal for vertical scaling, i.e.,** increasing the database's capacity by adding more resources (CPU, memory, storage) to a single machine.

You have the choice of using a SQL or NoSQL database

Database Options

2. **NoSQL databases**

- Data is not stored in relations (although the tabular form exists)
- There **is no predefined schema** (a flexible schema is maintained)
  - Data is not highly structured: unstructured or semi-structured data may be stored.
- Flexibility of schema speeds up development, as the developer has direct control over the structure of the database: "***data becomes like code"***.
- NoSQL databases could be: document stores, key-value stores, column stores or graph-base.
  - DynamoDB is a key-value store – meaning, they use hashing as their principal storage strategy.

- **Naturally ideal for horizontal scaling**, i.e., increasing the database's capacity by adding more machines and distributing data across them.

The Rise of NoSQL Databases

- Beginning in the **early 2000s**, web-based applications increasingly needed to handle massive amounts of data and traffic.
- **Scalability became crucial**:
  - Load can increase rapidly and unpredictably
  - Large servers are expensive and have physical limits
- The solution was to **move away from single large machines** and instead use clusters of small, commodity servers:
  - Data is fragmented (sharded) and replicated
  - Lower cost compared to vertical scaling
  - Greater overall reliability and fault tolerance
  - Natural fit for cloud-based infrastructure and storage

System Scaling

- Scalability is the property of a system to **handle a growing amount of work by adding resources** to the system.
  - From a database perspective: lots of data needs to be stored at high speed
- The point at which an application can no longer handle additional requests effectively is the limit of its scalability
- This limit is reached when a critical hardware resource runs out

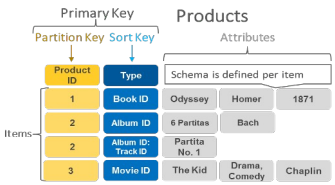
Horizontal Scaling of Databases

- Data needs to be divided and distributed across multiple machines.
- When queried for data, the system needs to know which machine stores what part of the data.
- The system should be able to:
  - map each individual record to a partition (e.g., a machine)
  - search for the record in a data structure within that partition
- This is the operation of a typical NoSQL database called a Key-Value Store

Amazon's DynamoDB:

An Example of a NoSQL Key-Value Store

- A key value store is a simple type of NoSQL database
- A key-value store stores data **as a collection of key-value pairs in which a key serves as a unique identifier**.
- Highly partitionable and allow horizontal scaling at scales that other types of databases cannot achieve.



An example table in DynamoDB

NoSQL Databases other than Key-Value Stores

Some major challenges faced by today's software systems regarding data include:

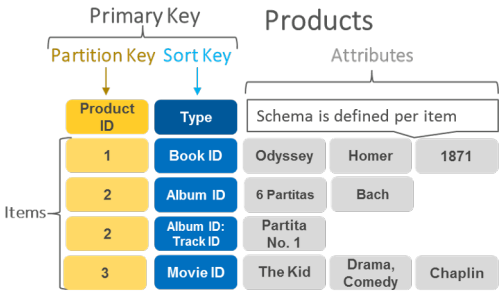
- Dealing with **big data**.
- Dealing with **unstructured data**.
- Dealing with **high speed data**.
- Having the ability to apply **data analytics, machine Learning** and deep Learning etc.
- Having the **ability to scale out horizontally**.
- Traditional relational database often aren't suited to serve these present day challenges.

Some NoSQL DB types:

Document databases (content management, APIs)
Wide-column stores (large-scale analytics)
Graph databases (social networks, recommendations)
Time-series databases (monitoring, IoT)
Search / index-oriented databases (full-text search, logs)

Amazon's DynamoDB, a key-value store

- Every 'row' in a 'table' is defined by a composite primary key:
  Primary key = (Partition key, Sort Key)
- A record with key (pk, sk), is assigned the partition *h*(pk), and sorted at *s*(sk), within that partition.
  - *h* is an internally maintained hash function.
  - *s* is an ordering function of an ordered data structure such as a height-balanced tree



A partition is an allocation of storage for a table, backed by solid state drives (SSDs) and automatically replicated across multiple Availability Zones within an AWS Region.

Creating a Table in DynamoDB (Boto3 Python)

```
import boto3

def create_movie_table(dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.create_table(
        TableName='Movies',
        KeySchema=[
            {
                'AttributeName': 'year',
                'KeyType': 'HASH'  # Partition key
            },
            {
                'AttributeName': 'title',
                'KeyType': 'RANGE'  # Sort key
            }
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'year',
                'AttributeType': 'N'
            },
            {
                'AttributeName': 'title',
                'AttributeType': 'S'
            },

        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 10,
            'WriteCapacityUnits': 10
        }
    )
    return table

if __name__ == '__main__':
    movie_table = create_movie_table()
    print("Table status:", movie_table.table_status)
```

Inserting an Item into a Table

```
from pprint import pprint
import boto3

def put_movie(title, year, plot, rating, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')
    response = table.put_item(
       Item={
            'year': year,
            'title': title,
            'info': {
                'plot': plot,
                'rating': rating
            }
        }
    )
    return response

if __name__ == '__main__':
    movie_resp = put_movie("The Big New Movie", 2015,
                           "Nothing happens at all.", 0)
    print("Put movie succeeded:")
    pprint(movie_resp, sort_dicts=False)
```

Reading an Item From a Table

```
def get_movie(title, year, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    try:
        response = table.get_item(Key={'year': year, 'title': title})
    except ClientError as e:
        print(e.response['Error']['Message'])
    else:
        return response['Item']


if __name__ == '__main__':
    movie = get_movie("The Big New Movie", 2015,)
    if movie:
        print("Get movie succeeded:")
        pprint(movie, sort_dicts=False)
```

Items can only be queried using their primary key (partition key, or both partition and sort keys)

In contrast, relational can query any combination of column names

Conditionally Deleting Items

```
def delete_underrated_movie(title, year, rating, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    try:
        response = table.delete_item(
            Key={
                'year': year,
                'title': title
            },
            ConditionExpression="info.rating <= :val",
            ExpressionAttributeValues={
                ":val": Decimal(rating)
            }
        )
    except ClientError as e:
        if e.response['Error']['Code'] == "ConditionalCheckFailedException":
            print(e.response['Error']['Message'])
        else:
            raise
    else:
        return response


if __name__ == '__main__':
    print("Attempting a conditional delete...")
    delete_response = delete_underrated_movie("The Big New Movie", 2015, 10)
    if delete_response:
        print("Delete movie succeeded:")
        pprint(delete_response)
```

Conditional expression

Similarly, items can be updated using table.update_item
(covered in the lab)

General Queries

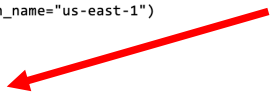
Get all movies released in a given year

```
import boto3
from boto3.dynamodb.conditions import Key

def query_movies(year, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name="us-east-1")

    table = dynamodb.Table('Movies')
    response = table.query(
        KeyConditionExpression=Key('year').eq(year)
    )
    return response['Items']


if __name__ == '__main__':
    query_year = 1985
    print(f"Movies from {query_year}")
    movies = query_movies(query_year)
    for movie in movies:
        print(movie['year'], ":", movie['title'])
```



General Queries

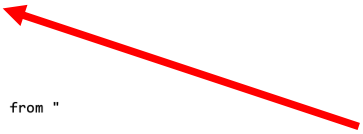
Get all movies released in a given year, with titles of a certain kind

```
def query_and_project_movies(year, title_range, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name="us-east-1")

    table = dynamodb.Table('Movies')
    print(f"Get year, title, genres, and lead actor")

    # Expression attribute names can only reference items in the projection expression.
    response = table.query(
        ProjectionExpression="#yr, title, info.genres, info.actors[0]",
        ExpressionAttributeNames={"#yr": "year"},
        KeyConditionExpression=
            Key('year').eq(year) & Key('title').between(title_range[0], title_range[1])
    )
    return response['Items']

if __name__ == '__main__':
    query_year = 1992
    query_range = ('A', 'L')
    print(f"Get movies from {query_year} with titles from "
          f"{query_range[0]} to {query_range[1]}")
    movies = query_and_project_movies(query_year, query_range)
    for movie in movies:
        print(f"\n{movie['year']} : {movie['title']}")
        pprint(movie['info'])
```



Identify the partition with the partition key, then, perform a range query using the sort key

Scanning: Querying Across Multiple Partitions

Print all movies between (and including) the years 1950 and 1959.

In a relational database this would be a straight-forward condition in a WHERE clause (as we see next week), as the entire table is available in one place.

In a key store, each key (i.e., the year) is mapped to a different partition, therefore all partitions in the range need to be **scanned**

Scan Queries

```
def scan_movies(year_range, display_movies, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    #scan and get the first page of results
    response = table.scan(FilterExpression=Key('year').between(year_range[0],
year_range[1]));
    data = response['Items']
    display_movies(data)

    #continue while there are more pages of results
    while 'LastEvaluatedKey' in response:
        response = table.scan(FilterExpression=Key('year').between(year_range[0],
year_range[1]), ExclusiveStartKey=response['LastEvaluatedKey'])
        data.extend(response['Items'])
        display_movies(data)

    return data

if __name__ == '__main__':
    def print_movies(movies):
        for movie in movies:
            #print(f"\n{movie['year']} : {movie['title']}")
            #pprint(movie['info'])
            pprint(movie)

    query_range = (1950, 1959)
    print(f"Scanning for movies released from {query_range[0]} to {query_range[1]}...")
    scan_movies(query_range, print_movies)
```

use in-built pagination to fetch and coalesce data from different partitions.

More Complex Querying and Table Joining

- Possible through scanning and additional code, but slow.
- Instead of implementing joins, choose table keys for queries corresponding to high-level functionalities
  - Tolerate redundancies and data duplication, etc.

a different philosophy to storage from relational databases where space is at a premium, redundancies are systematically removed, and a powerful language allows complex combinations of tables that are not designed for specific queries rather 'model the world'

Possible Database Use-cases for Your System

- **General application data**
  - Store core application data such as users, content, or domain-specific entities needed for normal operation
- **Data collection and history**
  - Store measurements, summaries, and historical records for monitoring, analysis, display through web interface etc.
- **System recovery and restart state**
  - Store checkpoints, last-known state, and progress markers so the system can resume correctly after crashes, restarts, or network failures
- **Configuration, preferences, and personalization**
  - Maintain system rules, user preferences, and adjustable parameters that shape application behaviour
- **Event tracking and notifications**
  - Record notable events (e.g. anomalies, milestones, interactions) and drive alerts or user-visible actions
- **Individual User activity and Data**
  - Track how the system is used over time to understand behaviour, performance, and trends
- **Shared state and coordination**
  - Maintain common state across components, users, or sessions to enable consistent system behaviour