Internet Of Things

Module 2

Wilgimol Thomas

Introduction to Fog Computing: Principles, Architectures, and Applications

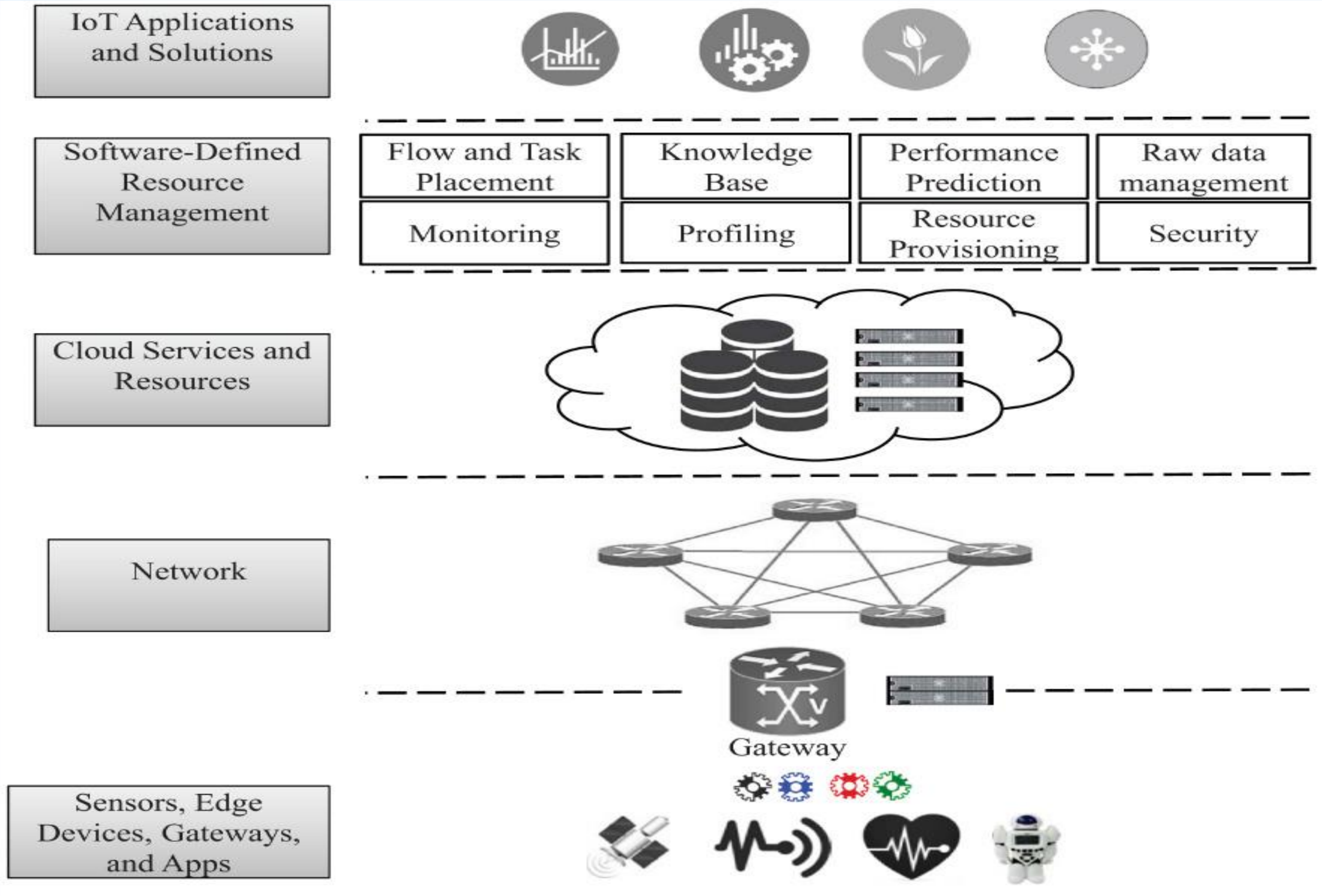
Principles:

- **What is Fog Computing?** Fog computing is a distributed computing model that extends cloud computing closer to the physical location of data sources (like IoT devices). It allows data to be processed locally instead of sending everything to a distant cloud server.
- **Why Use Fog Computing?** It reduces the time (latency) it takes for data to travel and get processed, which is crucial for real-time applications. It also saves bandwidth by processing some data locally.

Architectures:

- **Edge Devices**: These are the sensors or devices that collect data (e.g., temperature sensors, cameras).
- **Fog Layer**: This is where local processing happens. It includes devices like routers or gateways that can handle some of the computation tasks, reducing the load on cloud servers.
- **Cloud Layer**: This is used for more complex processing, storage, and analysis that can't be handled locally.

Fog Computing Reference Architecture



1. IoT Applications and Solutions:

- **Top Layer**: This layer represents the various Internet of Things (IoT) applications that utilize the underlying fog and cloud resources. These applications could range from smart agriculture and smart cities to health monitoring and industrial automation. The icons depict different domains, including data analytics, machine automation, and environmental monitoring.

2. Software-Defined Resource Management:

- **Second Layer**: This layer is responsible for managing and optimizing the resources in the fog environment. It includes:
  - **Flow and Task Placement**: Decides where tasks should be executed—either on edge devices, fog nodes, or cloud resources—to optimize performance and resource usage.
  - **Monitoring**: Constantly observes the status of resources, tasks, and network conditions.
  - **Knowledge Base**: Stores historical data and learned behaviors for better decision-making.
  - **Profiling**: Analyzes application and device characteristics to optimize resource allocation.
  - **Performance Prediction**: Predicts the performance of tasks based on current resource availability and network conditions.
  - **Resource Provisioning**: Allocates resources dynamically based on the demands of IoT applications.
  - **Raw Data Management**: Handles the preprocessing and storage of raw data generated by IoT devices.
  - **Security**: Ensures the protection of data, applications, and resources in the fog network.

3. Cloud Services and Resources:

- **Third Layer**: This layer represents traditional cloud services where massive data storage, processing power, and complex analytics are performed. Although cloud resources are powerful, the fog layer reduces latency by handling time-sensitive tasks locally before sending data to the cloud for further processing or storage.

4. Network:

- **Fourth Layer**: The network layer connects all components within the fog architecture. It ensures communication between sensors, edge devices, gateways, and cloud services. It includes switches, routers, and other networking hardware that facilitate data transfer across the architecture.

5. Sensors, Edge Devices, Gateways, and Apps:

- **Bottom Layer**: This is where data generation occurs. It includes:
  - **Sensors and Edge Devices**: Devices like cameras, temperature sensors, and actuators that generate data and perform initial processing tasks.
  - **Gateways**: Act as intermediaries between edge devices and the higher layers, handling tasks like protocol conversion, data filtering, and preliminary data analysis.
  - **Applications**: These could be mobile apps or embedded systems that directly interact with the sensors and edge devices.

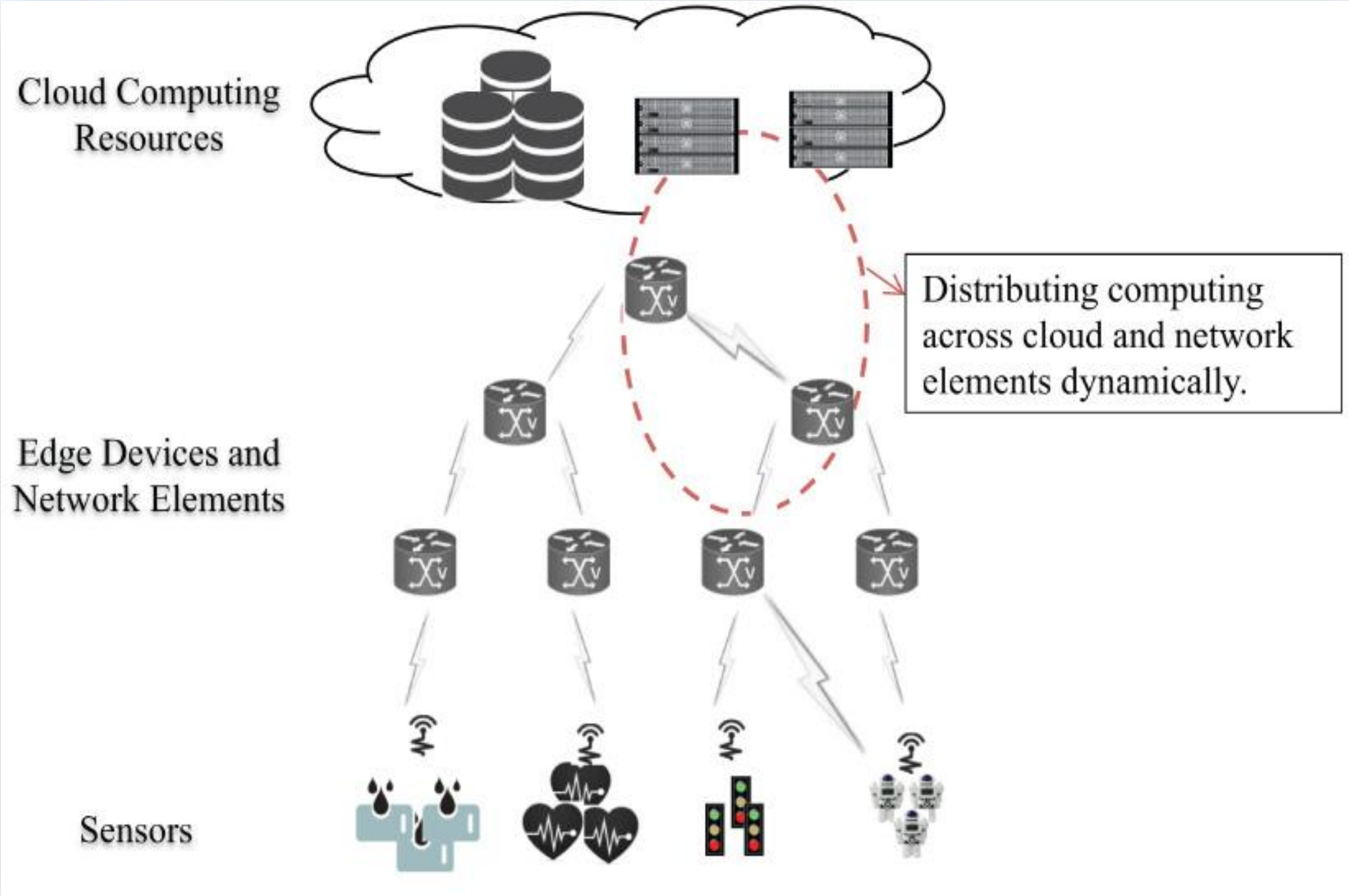
Applications:

- **Healthcare**: For example, a fog computing system can monitor patients in real-time, immediately analyzing critical data like heart rates without relying on a cloud server.
- **Augmented Reality**: Fog computing helps provide quick responses to user inputs, making the experience smooth and interactive.
- **Caching and Preprocessing**: Fog computing can store frequently accessed data locally, making web pages load faster and reducing the load on the internet.

(What are the advantages associated with Fog computing?)

Advantages of Fog Computing

- **Reduction of Network Traffic**: Fog computing processes data closer to its source, reducing the amount of data sent to the cloud.
- **Suitability for IoT Tasks**: Local processing of IoT data minimizes the need for cloud interaction, especially for location-specific queries.
- **Low Latency**: Ideal for real-time applications like robotics or autonomous vehicles where delays in cloud processing can be critical.
- **Scalability**: By processing data locally, it alleviates the cloud's burden, making the system more scalable



Distributing computing across cloud and network elements dynamically.

TinyOS and NesC

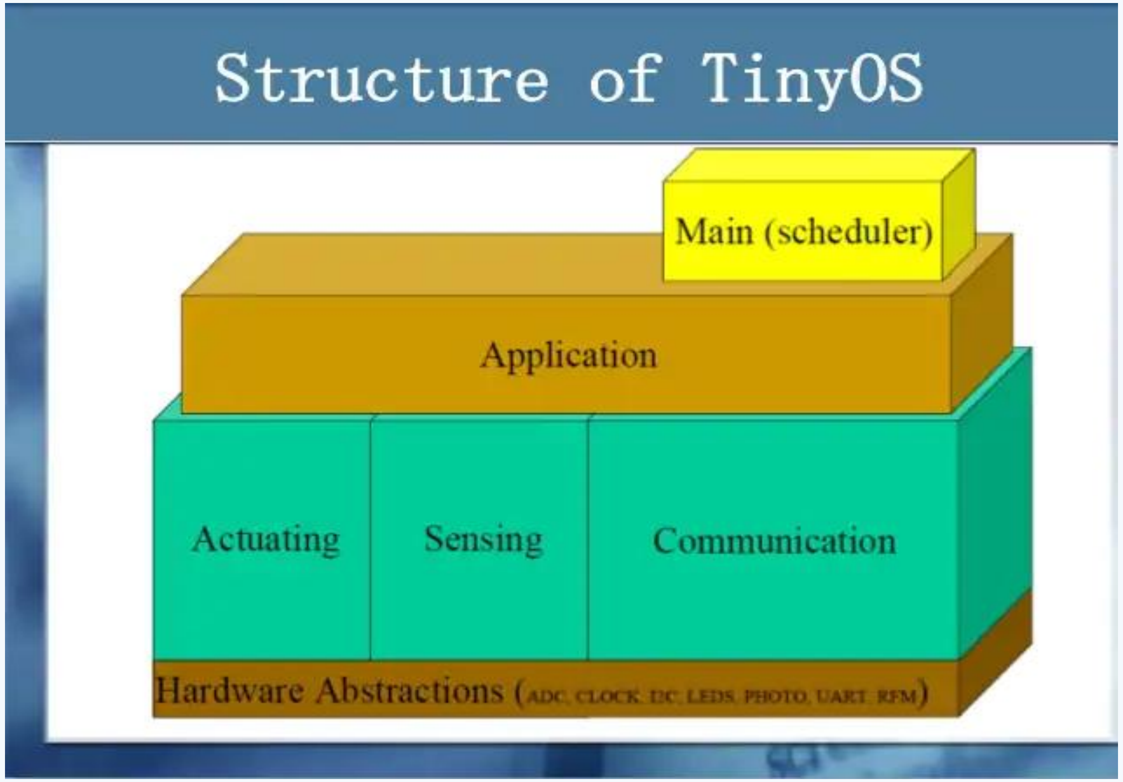
TinyOS Overview:

- **What is TinyOS?**
  - TinyOS is an open-source operating system designed specifically for low-power wireless devices, commonly used in sensor networks.
  - It is highly modular and component-based, which makes it lightweight and suitable for the resource constraints of tiny devices.
- **Key Features:**
  - **Event-Driven Execution Model:** TinyOS operates based on events, meaning that it reacts to external inputs like sensor readings rather than running continuously. This conserves energy.
  - **Concurrency:** It uses a non-preemptive scheduler that allows it to manage multiple tasks without the complexity of multi-threading, which would be too resource-intensive for tiny devices.
  - **Modular Architecture:** TinyOS is built from reusable components that can be combined in various ways to create different applications. This modularity is essential for adapting the OS to different hardware platforms and use cases.
  - **Low Power Consumption:** TinyOS is designed to minimize energy usage, which is critical for battery-powered devices that need to last for extended periods.

(Comment on the four broad requirements that motivate the design of TinyOS)

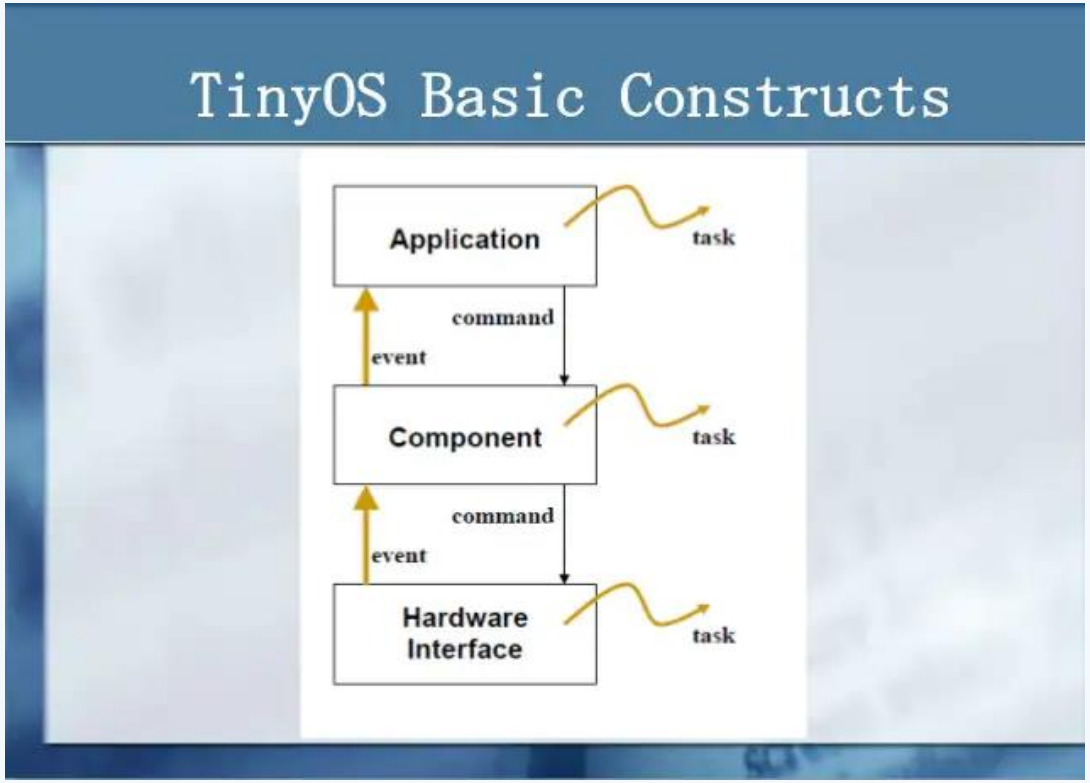
Four Broad Requirements that Motivate the Design of TinyOS:

1. **Resource Efficiency:** TinyOS needs to operate on devices with extremely limited resources (memory, CPU power, and battery). The OS is optimized to use as few resources as possible.
2. **Flexibility:** The modular design allows TinyOS to be adapted to various types of hardware and applications. Developers can select only the components they need, avoiding unnecessary overhead.
3. **Scalability:** TinyOS must scale from small networks of a few sensors to large networks with thousands of nodes, while still maintaining performance.
4. **Robustness and Reliability:** Given the environments where TinyOS is often deployed (e.g., outdoor sensor networks), the OS must be robust and reliable, able to recover from failures without human intervention.



Structure of TinyOS

Main (scheduler)
Application
Actuating
Sensing
Communication
Hardware Abstractions (ADC, CLOCK, I2C, LEDS, PHOTO, UART, RFM)



TinyOS Basic Constructs

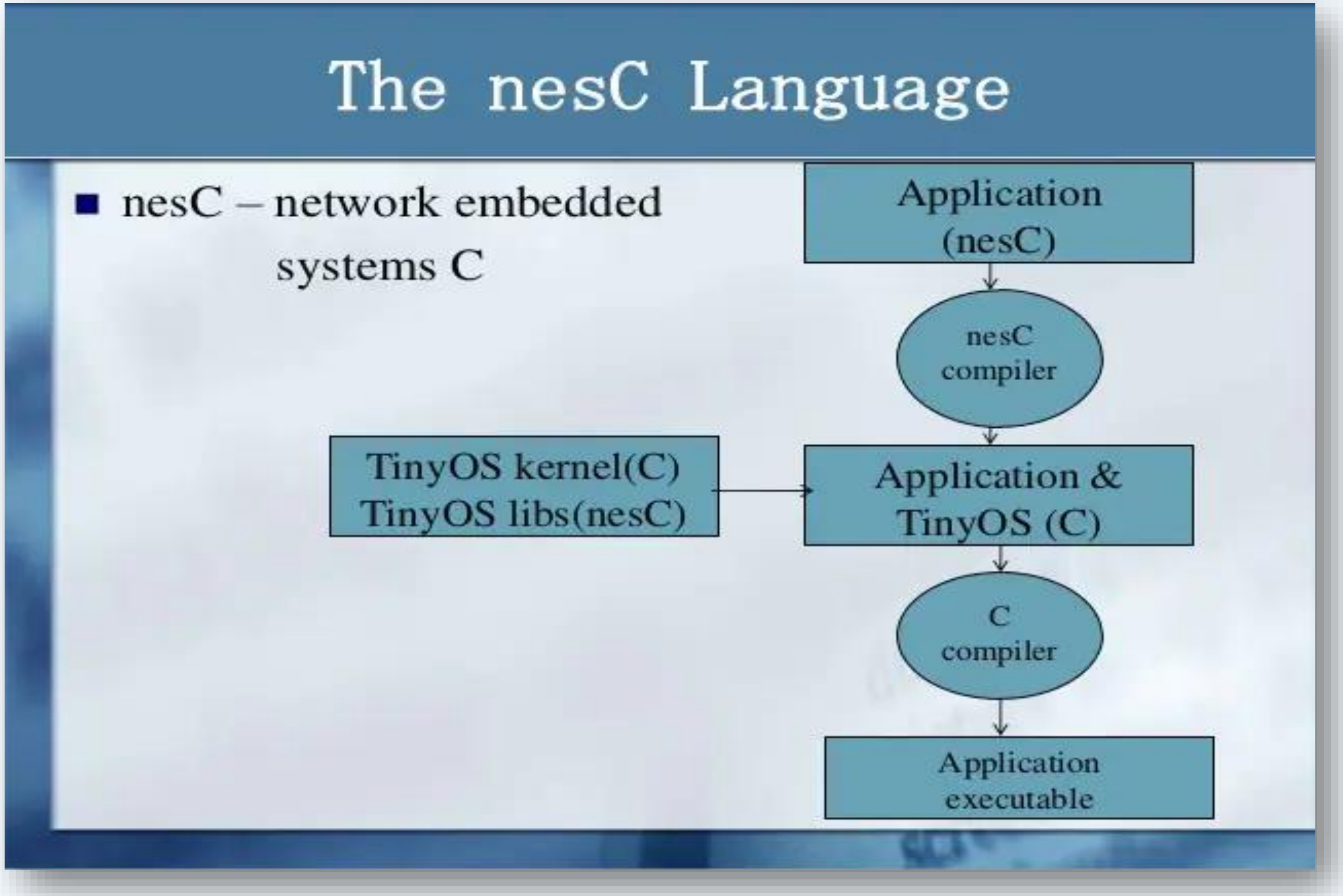
NesC Language:

- **What is NesC?**
  - NesC (Network embedded systems C) is a programming language specifically designed for developing applications on TinyOS.
  - It is an extension of the C programming language, adapted to support the modular and event-driven architecture of TinyOS.
- **Key Features:**
  - **Component-Based Design:** NesC programs are made up of components that interact with each other through well-defined interfaces. Each component in NesC can be either a module (which contains the logic) or a configuration (which connects modules together).
  - **Concurrency Model:** NesC supports TinyOS's concurrency model, where tasks are atomic and cannot be interrupted by other tasks, but events can preempt tasks. This approach simplifies programming by avoiding complex concurrency issues like race conditions.
  - **Static Memory Allocation:** NesC avoids dynamic memory allocation (which is common in traditional C programming) to reduce the risk of memory leaks and fragmentation. All memory is allocated at compile time, making the system more predictable and stable.
  - **Type Safety:** NesC includes features to catch common programming errors, such as data races, at compile time rather than at runtime, leading to more reliable code.

(Describe the design decisions for nesC.)

Design Decisions for NesC:

- **Concurrency Handling:** NesC uses tasks and events to manage concurrency, where tasks are non-preemptive and events can preempt tasks. This ensures efficient and predictable execution on resource-constrained devices.
- **Component Composition:** The use of components in NesC allows for code reuse and easier debugging. Components can be independently tested and then combined to form larger applications.
- **Static Analysis:** By focusing on compile-time checks, NesC ensures that many common programming errors are caught early, which is crucial for systems that must operate autonomously for long periods.



The nesC Language

- nesC – network embedded systems C

Application (nesC)

nesC compiler

TinyOS kernel(C)
TinyOS libs(nesC)

Application & TinyOS (C)

C compiler

Application executable

Programming frameworks for Internet of Things

The **Programming Frameworks for IoT (Internet of Things)** are tools and platforms designed to help developers build applications for IoT devices. IoT devices are typically small, with limited storage and processing power, and often rely on cloud-based systems for data storage, processing, and interaction with users.

1. **IoT and Cloud Computing**:
   - IoT devices often can't process everything on their own, so they use cloud systems to handle data storage, analysis, and other heavy tasks.
   - This combination allows IoT to extend the power of cloud computing to the real world, enabling large-scale data storage, powerful computing, and new services like "Sensing as a Service."
2. **Challenges in IoT Programming**:
   - **Heterogeneity**: IoT devices vary widely in terms of hardware, software, and communication methods.
   - **Data Volume**: IoT devices generate a huge amount of data in different formats and at different speeds.
   - **Latency Sensitivity**: Some applications, like traffic management, require fast data processing near the device (edge processing) to avoid delays.
   - **Device Availability**: IoT devices might frequently go offline due to limited power (batteries) or mobility, making consistent communication challenging.

3. Embedded Programming Languages:

- **C Language**: Most embedded IoT projects use C because of its performance, small memory footprint, and ability to interact directly with hardware.
- **nesC**: A variant of C used for sensor networks, focusing on optimizing reliability and resource use.
- **Keil C**: Adds specific features for embedded programming, like different memory models to optimize storage.
- **Dynamic C**: Supports advanced features like multitasking and memory management, tailored for embedded systems.
- **B#**: A multithreaded language inspired by C, C++, and Java, designed for resource-constrained systems like IoT devices.

4. Communication Protocols:

- **Remote Procedure Calls (RPC)**: Enables communication between devices and is useful for IoT, but might face challenges with interoperability.
- **REST (Representational State Transfer)**: Simplifies web-based interactions, making it easier to manage IoT devices via standard web tools.
- **CoAP (Constrained Application Protocol)**: A lightweight protocol optimized for the limited power and processing capabilities of IoT devices, making it efficient for IoT communication.

Communication Paradigms and Technologies in Resource-Constrained Environments

(Summarize the communication paradigms and technologies used in resource-constrained environments.)

Remote Procedure Call (RPC):

- **Definition**: A protocol that allows a program to execute a procedure (subroutine) on a remote server as if it were a local call.
- **Application**: Used in IoT for communication between devices, but traditional RPCs may face challenges with interoperability in heterogeneous IoT environments.
- **Variants**: Lightweight versions like Embedded RPC (ERPC) and S-RPC are optimized for resource-constrained environments by minimizing the overhead of encoding/decoding and data buffering.

1. **Representational State Transfer (REST)**:
   - **Definition**: A web-based communication protocol that simplifies web application development by using standard web technologies.
   - **Application**: REST is widely used in IoT for client-server communication due to its simplicity and scalability. It supports stateless interactions and caching, making it efficient for web-enabled IoT devices.
   - **Extensions**: CREST (Computational REST) extends REST for computational exchanges, supporting more complex IoT interactions where computation rather than just data transfer is needed.

3. **Constrained Application Protocol (CoAP)**:

   - **Definition**: A specialized web transfer protocol designed for use with constrained devices and networks, optimized for low-power, low-bandwidth environments.
   - **Application**: CoAP is similar to HTTP but designed for IoT, using UDP for asynchronous message exchanges, reducing communication overhead.
   - **Features**: CoAP supports four types of messages (Confirmable, Non-confirmable, Acknowledgement, Reset) and methods similar to HTTP (GET, POST, PUT, DELETE). It is particularly useful for scenarios requiring reliable communication with minimal power consumption.

(Outline the major challenges faced in the Fog paradigm.)

Challenges in the Fog Paradigm

- **Data Security and Privacy**: Ensuring secure data processing at the edge.
- **Resource Management**: Efficient allocation of processing power between cloud and fog layers.
- **Scalability**: Managing an increasing number of edge devices.
- **Interoperability**: Ensuring different devices and networks can communicate effectively.

EXISTING IoT FRAMEWORKS

Existing IoT frameworks are tools and platforms designed to simplify the development of IoT applications by providing predefined structures and functionalities. Here's a simplified explanation of the key IoT frameworks mentioned in your PPT:

1. Mobile Fog

- **Concept**: Extends fog computing, which is an architecture that brings cloud computing capabilities closer to IoT devices (near the "edge" of the network).
- **Features**: Mobile Fog allows for low-latency processing at the edge while more complex tasks are handled in the cloud. It provides a Platform as a Service (PaaS) model, enabling dynamic scaling and managing geographically distributed IoT processes.

2. ELIoT (Erlang Language for IoT)

- **Concept**: A lightweight version of Erlang, a programming language designed for real-time systems, optimized for IoT applications.
- **Features**: ELIoT strips down the standard Erlang to remove unnecessary components for IoT, making it suitable for resource-constrained devices. It offers a small library for decentralized systems, an interpreter for IoT devices, and a simulator for testing.

3. Compose API

- **Concept**: A service platform for IoT that uses RESTful APIs to allow devices and users to interact over the web.
- **Features**: It turns IoT devices into "Web Objects" that can be managed and interacted with using standard web protocols. The framework simplifies programming by exposing IoT devices through APIs, which can be composed to create complex services.

4. Distributed Dataflow Support for IoT

- **Concept**: Extends existing IoT platforms to support distributed dataflows, where data processing tasks are spread across multiple devices.
- **Features**: The framework focuses on three key aspects: flow ownership, naming of nodes, and distinguishing local from remote connections. It allows for parallel processing of data, enabling efficient distribution of tasks in an IoT network.

5. PyoT

- **Concept**: A programming framework for Wireless Sensor Networks (WSNs) that communicates via the Internet using protocols like 6LoWPAN and CoAP.
- **Features**: PyoT abstracts sensors and actuators as software objects that can be managed using Python. It supports in-network processing, where part of the application logic is executed directly on the devices, reducing the load on central servers.

6. Dripcast

- **Concept**: A Java-based framework that integrates smart devices with cloud computing, making Java objects available across the network.
- **Features**: Dripcast uses Java Remote Procedure Calls (RPC) to manage and process objects in the cloud. It enables transparent interaction with cloud-based Java objects, facilitating IoT application development.

7. Calvin

- **Concept**: Merges IoT and cloud computing into a unified model, based on the actor-model and flow-based computing.
- **Features**: Calvin simplifies development through four phases: Describe, Connect, Deploy, and Manage. It allows actors (devices, services, etc.) to communicate and migrate across different runtime environments, promoting scalability and flexibility.

8. Simurgh

- **Concept**: A high-level framework for developing and managing IoT services, exposing them as RESTful APIs.

Features: Simurgh includes components for discovering, composing, and executing IoT services. It provides tools to manage IoT services and reuse flow patterns, aiding developers in building complex IoT applications.

9. High-Level Application Development for IoT

- **Concept**: Proposes a detailed framework for developing IoT applications by focusing on different areas of concern (domain-specific, functional, deployment, and platform-specific).
- **Features**: The framework uses modeling languages and automation techniques to support the development process, making it easier to manage the complexity of IoT systems.

10. PatRICIA

- **Concept**: An IoT framework for cloud platforms based on an "intent"-based programming model.
- **Features**: Developers specify their "intent" (what they want to monitor or control), and the framework manages the execution. This model simplifies IoT programming by hiding the complexity of underlying device interactions.

Coordination Languages

What are Coordination Languages?

- **Definition:** Coordination languages are specialized programming languages that are used to manage and control the interactions between different components or processes in a parallel or distributed computing environment. They focus on how different parts of a system communicate and work together rather than on the computations themselves.

Key Features of Coordination Languages:

- **Separation of Concerns:** Coordination languages separate the coordination logic (how components interact) from the computation logic (what each component does). This makes it easier to manage complex systems by focusing on the communication between components.
- **Parallelism and Concurrency:** They are designed to handle tasks that need to run simultaneously (in parallel) or tasks that require coordination between multiple concurrent processes. This makes them ideal for distributed systems and multi-core processors.
- **Communication:** These languages provide mechanisms for components to communicate, share data, and synchronize their operations. They ensure that different parts of a system can work together effectively, even if they are running on different machines or processors.

(List the features in coordination languages - Linda, eLinda, Orc, and Jolie)

Examples of Coordination Languages:

1. **Linda:**
   - **Features:** Linda uses a shared memory space known as a "tuple space" where processes can write, read, and take tuples (data objects). It focuses on generative communication, meaning once a tuple is generated and placed in the tuple space, it can be accessed independently by other processes.
   - **Use Case:** It’s often used in parallel computing environments where multiple processes need to communicate and synchronize their activities.
2. **eLinda:**
   - **Features:** eLinda is an extension of Linda with additional features like enhanced tuple operations and better support for distributed computing. It aims to improve efficiency and scalability in distributed systems.
   - **Use Case:** Useful in more complex distributed systems that require sophisticated coordination and communication mechanisms.

4. **Orc:**

   - **Features:** Orc (Orchestration) is designed to manage the execution of web services and other distributed resources. It allows the coordination of multiple tasks, handles failures gracefully, and ensures that tasks are completed in the right order.
   - **Use Case:** Typically used in systems that need to orchestrate multiple web services or distributed components, ensuring they work together smoothly.

5. **Jolie:**

   - **Features:** Jolie is a service-oriented coordination language that focuses on the composition and orchestration of services. It supports communication over various protocols (like HTTP, SOAP) and allows for flexible service interactions.
   - **Use Case:** Ideal for building and coordinating microservices, where different services need to be integrated and managed.

Polyglot Programming

(Explain Polyglot Programming.)

What is Polyglot Programming?

- **Definition:** Polyglot programming refers to the practice of using multiple programming languages within a single application or project. Instead of relying on just one language, developers choose the best language for each specific task or component.

Why Use Polyglot Programming?

- **Best Tool for the Job:** Different programming languages have strengths and weaknesses. By using multiple languages, you can leverage the strengths of each one. For example, you might use Python for data analysis, Java for building scalable backend services, and JavaScript for interactive web interfaces.
- **Flexibility:** It allows developers to be more flexible and choose the most efficient, powerful, or convenient language depending on the problem they are solving.
- **Innovation:** It encourages innovation because developers aren’t limited by the constraints of a single language. They can experiment with new languages and technologies.

Common Examples of Polyglot Programming:

- **Web Development:** A common example is in web development, where HTML, CSS, and JavaScript are used together for the frontend, while the backend might be developed in Python, Java, Ruby, or any other language that fits the needs of the project.
- **Microservices:** In a microservices architecture, different services can be written in different languages. This allows each service to be developed using the language that is most appropriate for its specific requirements.

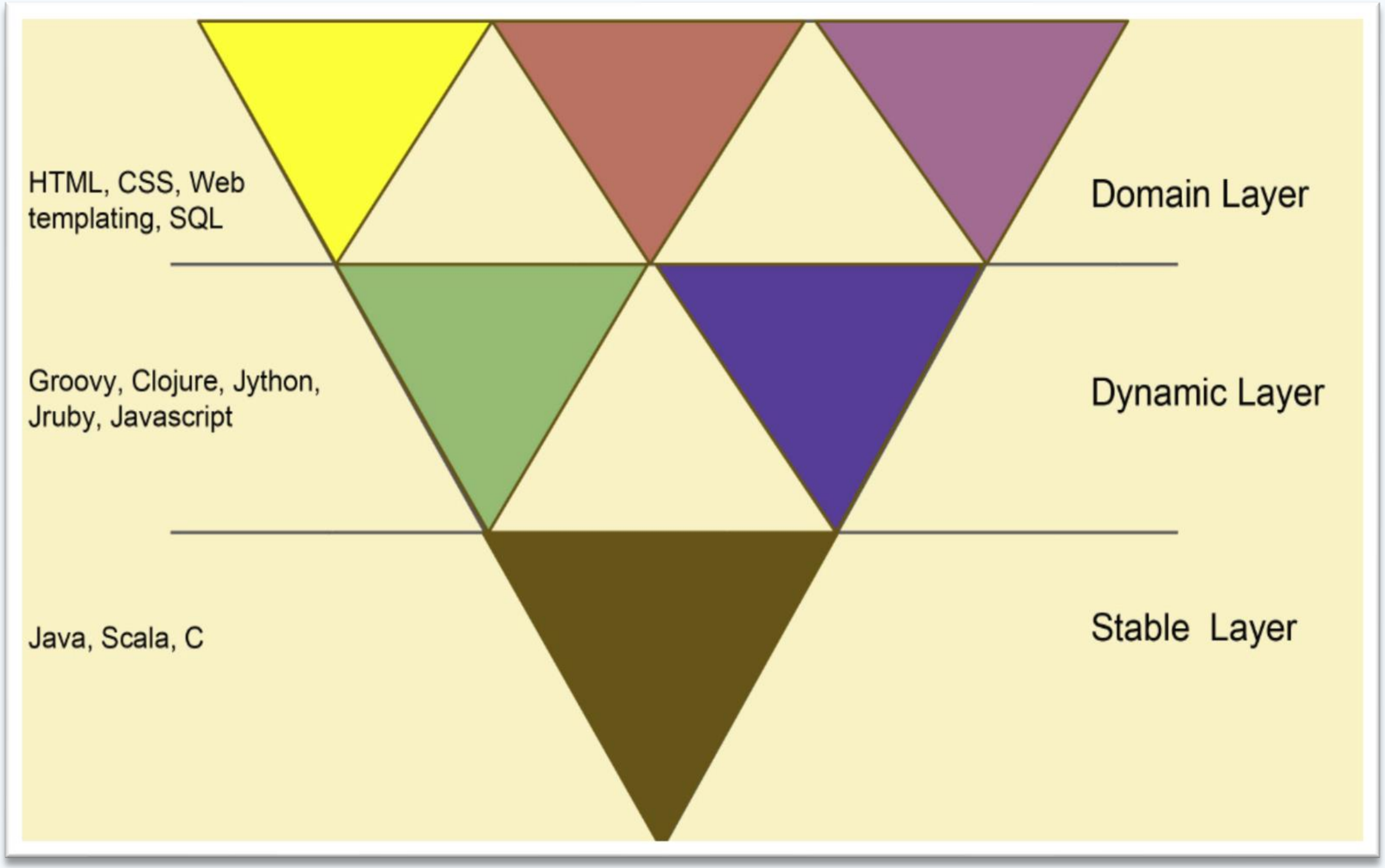
Challenges of Polyglot Programming:

- **Complexity:** Managing multiple languages can increase the complexity of the development process. Developers need to be familiar with different programming paradigms and tools.
- **Integration:** Ensuring that different languages and their respective components work together smoothly can be challenging. It requires careful planning and often additional tools or frameworks for integration.
- **Maintenance:** Maintaining a polyglot system might be more difficult because of the need to understand and update code written in multiple languages.

Conclusion:

- Polyglot programming is a powerful approach that enables developers to use the best tools for each part of a project. While it offers many advantages, it also comes with challenges that need to be managed carefully to ensure a successful outcome.

Inverse Pyramid for Polyglot Programming


HTML, CSS, Web templating, SQL
Domain Layer
Groovy, Clojure, Jython, Jruby, Javascript
Dynamic Layer
Java, Scala, C
Stable Layer

IoT PROGRAMMING APPROACHES

IoT Programming Approaches refer to the different methods used to develop applications for Internet of Things (IoT) systems. Each approach has its strengths and is chosen based on the specific needs of the application. Here's a simplified explanation of the key IoT programming approaches:

1. Node-Centric Programming:

- **Focus**: Each IoT device (or node) is individually programmed to handle tasks like communication, data collection, and controlling other devices.
- **Advantages**: Offers full control over how each device operates.
- **Challenges**: It's labor-intensive and doesn't easily allow for reusing code across different devices. It can also make scaling and maintenance difficult.

2. Database Approach:

- **Focus**: Treats each IoT device as part of a virtual database, where developers can issue queries to collect data from sensor nodes.
- **Advantages**: Simplifies data collection by treating sensor data as if it's coming from a database.
- **Challenges**: Doesn't support complex application logic at the device level, which limits its usefulness for more advanced IoT applications.

3. Macro Programming:

- **Focus**: Allows developers to write high-level code that abstracts away the low-level details, making it easier to specify what the entire network of IoT devices should do.
- **Advantages**: Supports modular and rapid development by hiding the complexity of low-level programming, making it easier to manage large-scale IoT systems.
- **Challenges**: Requires learning the abstractions and tools provided by the macro programming environment.

4. Model-Driven Development:

- **Focus**: Uses models to represent different aspects of the system, such as functionality, deployment, and hardware interaction, to separate concerns and reduce complexity.
- **Advantages**: Makes it easier to manage complexity by allowing developers to focus on specific parts of the system at a time (e.g., logic vs. deployment). It promotes reusability and simplifies the development process.
- **Challenges**: Can be complex to set up and may require specialized tools and training.