PROGRAMMING FRAMEWORKS FOR INTERNET OF THINGS

Prepared By,

SHELLY SHIJU GEORGE

ASSISTANT PROFESSOR

INTRODUCTION

- IoT devices are generally characterized as small things in the real world with limited storage and processing capacity, which may not be capable of processing a complete computing activity by themselves.
- They may need the computational capabilities of Cloud-based back-ends to complete the processing tasks and web-based front-ends to interact with the user.
- The Cloud infrastructure complements the things, by supporting device **virtualization, availability, provisioning of resources, data storage, and data analytics.**

INTRODUCTION (CONTINUED)

- The IoT by its nature will extend the scope of Cloud computing to the real world in a more distributed and dynamic way.
- IoT with Cloud will create new avenues for computing: huge storage capacity for IoT data in cloud; massive computing capabilities to collect, analyze, process, and archive those data; and new platforms such as SaaS (Sensing as a Service), SAaaS (Sensing and Actuation as a Service), and VSaaS (Video Surveillance as a Service) will open up to users.

INTRODUCTION (CONTINUED)

- Heterogeneity and the volume of data generated are two of the biggest concerns.
- Heterogeneity spans through hardware, software, and communication platforms.
- The data generated from these devices are generally in huge volume, are in various forms, and are generated at varying speeds.
- Since IoT applications will be distributed over a wide and varying geographical area, support for corrective and evolutionary maintenance of applications will determine the feasibility of application deployment.

INTRODUCTION (CONTINUED)

- Further, some of the IoT applications such as traffic management will be latency-sensitive, and this warrants edge-processing support by the programming framework.
- Another difficulty to face when programming IoT is how to cope with frequent periods of nonavailability of devices, caused by mobilities and limited energy supplies from batteries.
- Developing a simplified programming model that can provide solutions for the above set of challenges will remain a continuous pursuit for the IoT community.

OVERVIEW

- During the lifecycle of IoT applications, the footprint of an application and the cost of its language runtime play a huge role in the sustainability of an application.
- C has been used predominantly in embedded applications development due to its performance; moreover, it can occupy the same position in IoT programming too.
- Further, the choice of communication protocols also has a huge implication in the cost of IoT applications on devices.
- Remote Procedure Calls (RPC), Representational state transfer (REST), and Constrained Application Protocol (CoAP) are some of the communication methods that are being currently incorporated into IoT communication stacks.

OVERVIEW (CONTINUED)

- A complete programming framework in a distributed environment requires not only a stable computing language such as C, but also a coordination language that can manage communications between various components of an IoT ecosystem.
- An explicit coordination language can tackle many of the challenges.
- It can manage communication between heterogeneous devices, coordinate interaction with the Cloud and devices, handle asynchronous data arrival, and can also provide support for fault tolerance.

OVERVIEW (CONTINUED)

- The method of using more than one language in a given application is known as polyglot programming.
- Polyglot programming is being widely used in web applications development, and it can provide the same advantages for IoT programming too.

EMBEDDED DEVICE PROGRAMMING LANGUAGES

- Although there are various programming languages in the embedded programming domain, the vast majority of projects, about 80%, are either implemented in C and its flavors, or in a combination of C and other languages such as C++.
- Some of the striking features of C that aid in embedded development are performance, small memory foot-print, access to low-level hardware, availability of a large number of trained/experienced C programmers, short learning curve, and compiler support for the vast majority of devices.

The ANSI C standard provides customized support for embedded programming. Many embedded C-compilers based on ANSI C usually:

1. support low-level coding to exploit the underlying hardware,
2. support in-line assembly code,
3. flag dynamic memory-allocation and recursion,
4. provide exclusive access to I/O registers,
5. support accessing registers through memory pointers, and
6. allow bit-level access.

nesC, Keil C, Dynamic C, and B# are some of the flavors of C used in embedded programming.

nesC

- nesC is a dialect of C that has been used predominantly in sensor-nodes programming.
- It was designed to implement TinyOS, an operating system for sensor networks.
- It is also used to develop embedded applications and libraries.
- In nesC, an application is a combination of scheduler and components wired together by specialized mapping constructs.
- nesC extends C through a set of new keywords.

nesC (CONTINUED)

- To improve reliability and optimization, nesC programs are subject to whole program analysis and optimization at compile time.
- nesC prohibits many features that hinder static analysis, such as function pointers and dynamic memory allocation.
- Since nesC programs will not have indirections, call-graph is known fully at compile time, aiding in optimized code generation.

Keil C

- Keil C is a widely used programming language for embedded devices.
- It has added some key features to ANSI C to make it more suitable for embedded device programming.
- To optimize storage requirements, three types of memory models are available for programmers: small, compact, and large.
- New keywords such as *alien, interrupt, bit, data, xdata, reentrant*, and so forth, are added to the traditional C keyword set.

Keil C (CONTINUED)

Keil C supports two types of pointers:

- generic pointers: can access any variable regardless of its location
- memory-specific pointers: can access variables stored in data memory

The memory-specific-pointers-based code execute faster than the equivalent code using generic pointers.

This is due to the fact that the compilers can optimize the memory access, since the memory area accessed by pointers is known at compile time.

Dynamic C

- Some key features in Dynamic C are function chaining and cooperative multitasking.
- Segments of code can be distributed in one or more functions through function chaining.
- Whenever a function chain executes, all the segments belonging to that particular chain execute.
- Function chains can be used to perform data initialization, data recovery, and other kinds of special tasks as desired by the programmer.
- The language provides two directives #*makechain*, #*funcchain* and a keyword *segchain* to manage and define function chains.

Dynamic C (CONTINUED)

#*makechain chain_name*: creates a function chain by the given name.

#*funcchain chain_name func_name[chain_name*]: Adds a function or another function chain to a function chain.

segchain chain_name {statements}: This is used for function-chain definitions. The program segment enclosed within curly brackets will be attached to the named function chain.

- The language stipulates *segchain* definitions to appear immediately after data declarations and before executable statements, as shown in the following code snippet.

Dynamic C (CONTINUED)

```
int foo(){
// data declarations
segchain recover{
// some statements which execute under function chain recover().
                    }
segchain chain_x{
// some statements which execute under function chain chain_x().
                    }
// function body of foo.

}

int foo1(){
// data declarations
segchain recover{
// some statements which execute under function chain recover().
                    }
// function body of foo1.

}
```

Dynamic C (CONTINUED)

Calling a function chain inside a program is similar to calling a void function that has no parameters.

```
int foo2(){
        .....
        .....
        recover()/* executes all the statements defined under
                    function chain recover */
      }
```

Dynamic C (CONTINUED)

- The order of execution of statements inside a function chain is not guaranteed.
- Dynamic C's *costate* statement provides support for cooperative multitasking.
- It provides multiple threads of control, through independent program counters that can be switched in between explicitly.
- The following code snippet is an example.

```
for(;;){
        costate{
                waitfor(tcp_packet_port_21());
                yield; // force context switch.
                ...
                }
        costate{
                waitfor(tcp_packet_port_23());
                }

        }
```

Dynamic C (CONTINUED)

- The *yield* statement immediately passes control to another *costate* segment.
- If the control returns to the first *costate* segment, then the execution resumes from the statement following the *yield* statement.
- Dynamic C also has keywords *shared* and *protected*, which support data that are shared between different contexts and are stored in battery-backed memory, respectively.

B#

- B# is a multithreaded programming language designed for constrained systems.
- Although C inspires it, its features are derived from a host of languages such as Java, C++, and C#.
- It supports object oriented programming.
- The idea of boxing/unboxing conversions is from C#. For example, a float value can be converted to an object and back to float, as shown in the following code snippet.

```
class test{
static void Main(){
      float         i = 123;
      object        obj = i;              // boxing
      float         j = (float)obj;       // Unboxing
      }
}
```

B# (CONTINUED)

- The field property is also similar to C#.
- B# provides support for multithreading and synchronization through *lock* and *start* statements, which are similar to when and *cobegin*, from Edison.
- lock provides mutual exclusion and synchronization support, whereas *start* is used to initiate threads.
- Other important features are device-addressing registers and explicit support for interrupt handlers.
- These features are directly supported by the underlying Embedded Virtual Machine (EVM), which interprets and executes the binary code generated by the B# assembler on a stack-based machine.

B# (CONTINUED)

- The B# EVM runs on a target architecture, thereby hiding the hardware nuances (distinctions) from the programmer.
- Presence of EVM promotes reusability of components.
- Also, since the EVM is based on the stack-machine model, the code size is much reduced.
- The EVM also has a small kernel for managing threads.

B# (CONTINUED)

- All of the previously described languages have been optimized for resource-constrained devices.
- While designing embedded programs, a measured choice on the flavor of C is quite an important decision from the viewpoint of an IoT programmer.
- An IoT programmer may not restrict himself or herself to a C-flavored language.
- Many other languages, such as C++, Java, and JavaScript have been stripped down to run on embedded devices.

MESSAGE PASSING IN DEVICES

- Some of the communication paradigms and technologies such as RPC, REST, and CoAP that can be used in resource-constrained environments.

RPC (REMOTE PROCEDURE CALL)

- RPC is an abstraction for procedural calls across languages, platforms, and protection mechanisms.
- For IoT, RPC can support communication between devices as it implements the request/response communication pattern.
- Typical RPC calls exhibit synchronistic behavior.
- When RPC messages are transported over the network, all the parameters are serialized into a sequence of bytes.
- Since serialization of primitive data types is a simple concatenation of individual bytes, the serialization of complex data structures and objects is often tightly coupled to platforms and programming languages.
- This strongly hinders the applicability of RPCs in IoT due to interoperability concerns.

RPC (CONTINUED)

- Lightweight Remote Procedure Call (LRPC) was designed for optimized communication between protection domains in the same machine, but not across machines.
- Embedded RPC (ERPC) in Marionette uses a fat client such as a PC and thin servers such as nodes architecture.
- This allows resource-rich clients to directly call functions on applications in embedded devices.
- It provides poke and peek commands that can be used on any variables in a node's heap.
- S-RPC is another lightweight remote procedure-call for heterogeneous WSN networks.

RPC (CONTINUED)

- S-RPC tries to minimize the resource requirements for encoding/decoding and data buffering.
- A trade-off is achieved based on the data types supported and their resource consumption.
- Also, a new data representation scheme is defined which minimizes the overhead on packets.
- A lightweight RPC has been incorporated into the TinyOS, nesC environment.
- This approach promises ease of use, lightweight implementation, local call-semantics, and adaptability.

REST

- The main aim of the REST was to simplify the web-application development and interaction.

It leverages on the tools available on the Internet and stipulates the following constraints on application development:

- Should be based on client-server architecture and the servers should be stateless
- Support should be provided for caching at the client side

REST (CONTINUED)

- The interface to servers should be generic and standardized (URI)
- Layering in the application architecture should be supported and each of the layers shall be independent
- Optional code-on demand should be extended to clients having the capability

REST (CONTINUED)

These constraints, combined with the following principles, define the RESTful approach to application development.

- Everything on the Internet is a resource
- Unique identifiers are available to identify the resources
- Generic and simple interfaces are available to work with those resources

REST (CONTINUED)

- Communication between client and servers can be through representation of resources
- Resource representation through sequence of bytes is followed by some metadata explaining the organization of the data
- Since transactions are stateless, all interactions should be context-free
- Layering is supported, and hence intermediaries should be transparent

REST (CONTINUED)

- Advantage of RESTful components is that they can be composed to produce mashups, giving rise to new components which are also RESTful.

Essential characteristic features of a composing language that can compose RESTful components together:

- Support for dynamic and late binding
- Uniform interface support for composed resource manipulation
- Support for dynamic typing
- Support for content-type negotiation
- Support for state inspection of compositions by the client

REST (CONTINUED)

- Although the uniform interface constraint promotes scalability by shifting the variability from interface to resource representation, it also narrows the focus of RESTful approaches to data and its representation.
- Also, in the Internet, the exchanges need not be limited to data and its representation; there can be more than just the pure data.
- For these cases, the optional code-on demand constraint for clients has been found to be inadequate for exchanges other than content.
- Also, the RESTful approach poses a challenge for those applications that require stateful interactions.

REST (CONTINUED)

- CREST (Computational REST) tries to address these problems.
- Here, the focus is on exchanges of computation rather than on data exchange.
- Instead of client-server nomenclature, everyone is addressed as peers; some may be strong and some may be weak, based upon the available computing power.
- Functional languages such as Scheme allow computations to be suspended at a point and encapsulated as a single entity to be resumed at a later point in time, through “continuation.”
- CREST’s focus is on these sorts of computation.

REST (CONTINUED)

- It supports the model of “computations stopping at a point in a node, exchanged with another node, resumed from the suspended point at the new node.”
- As said earlier, both the nodes are peers.

CREST has some principles along the lines of REST:

- All computations are resources and are uniquely identified
- Representation of resources through expressions and metadata
- All computations are context-free

REST (CONTINUED)

- Support for layering and transparent intermediaries
- All the computations should be included inside HTTP
- Computations can produce different results at different times
- Between calls they can maintain states that may aid computations such as aggregation
- Between different calls, computations should support independence
- Parallel synchronous invocations should not corrupt data

Computations on a peer or on different peers can be composed to create mashups. Peers can share the load of computations to promote scaling and latency-sensitive applications.

CoAP

- Since HTTP/TCP stack is known to be resource demanding on constrained devices, protocols such as Embedded Binary HTTP (EBHTTP) and Compressed HTTP Over PAN (CHoPAN) have been proposed.
- However, the issue of reliable communications still remains a concern.
- The IETF work group, Constrained RESTful Environments (CoRE), has developed a new web-transfer protocol called **Constrained Application Protocol** (CoAP), which is optimized for constrained power and processing capabilities of IoT.
- Although the protocol is still under standardization, various implementations are in use.
- CoAP in simpler terms is a two-layered protocol: a messages layer, interacting with the UDP, and another layer for request/response interactions using methods and response code, as done in HTTP.
- In contrast to HTTP, CoAP exchanges messages asynchronously and uses UDP.

CoAP (CONTINUED)

- The CoAP has four types of messages: ***Acknowledgement, Reset, Confirmable (CON), and Non-Confirmable (NON).***
- The non-confirmable messages are used to allow sending requests that may not require reliability.
- Reliability is provided by the message layer and will be activated when Confirmable messages are used.
- The Request methods are: GET, POST, PUT, and DELETE of HTTP.
- CoAP has been implemented on Contiki, which is an operating system for sensor networks and in TinyOS as Tiny-CoAP.

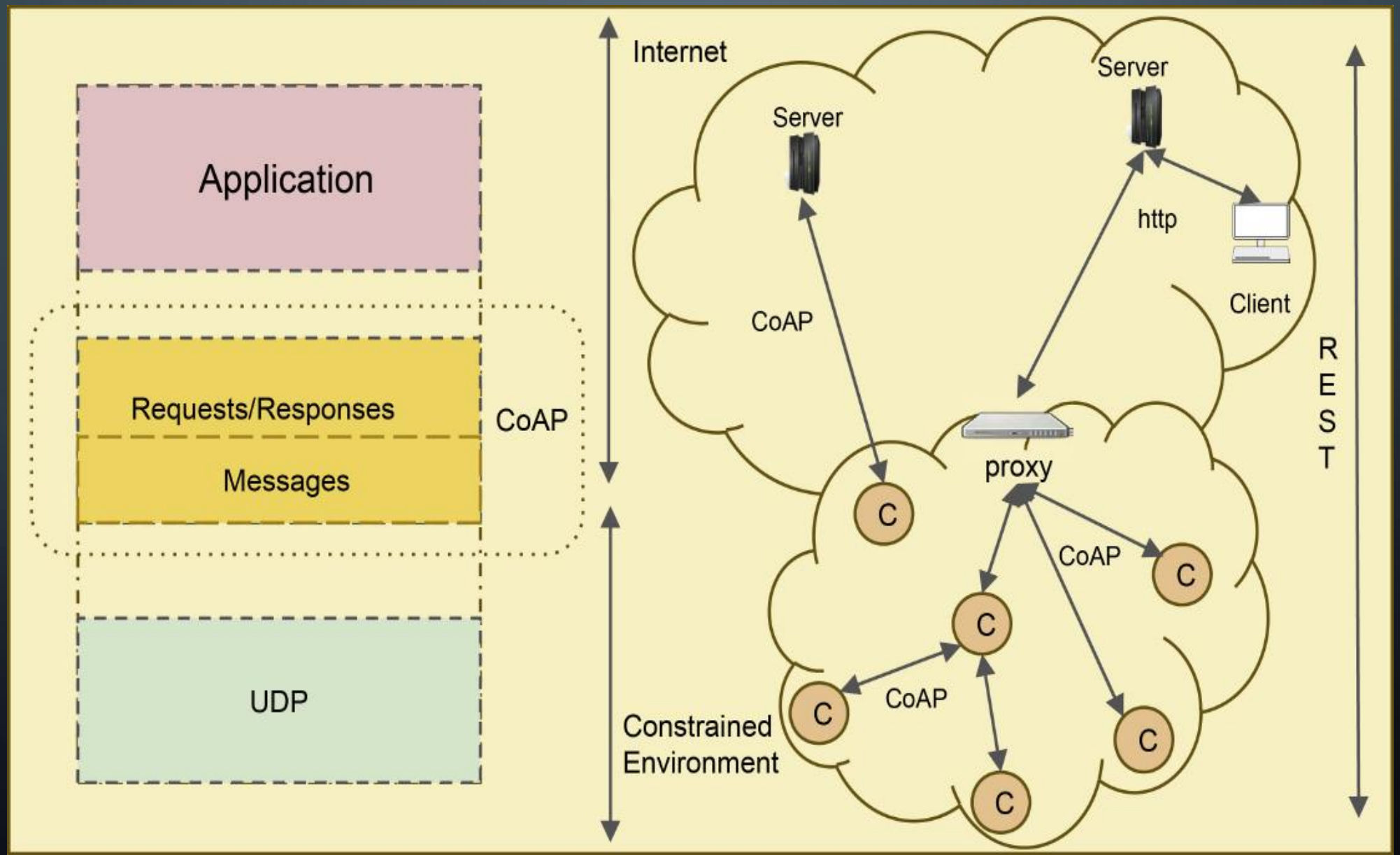
CoAP (CONTINUED)

- Many approaches have been used to evaluate the performance of CoAP.
- A Total Cost of Ownership (TCO) model for applications in a constrained environment has been used to compare HTTP versus CoAP.

The major observations from the comparison are as follows:

- CoAP is more efficient for applications on smart objects, engaged in frequent communication sessions
- CoAP is cost-effective whenever the battery/power-source replacements prove costly
- Whenever the charges for the data communication are volume-based, CoAP is found to be more cost-effective
- CoAP has been found to be more beneficial cost-wise in push mode than in pull mode

CoAP LAYERS AND INTEGRATION OF CONSTRAINED DEVICES WITH THE INTERNET



CoAP (CONTINUED)

For IoT/CoT, the advantages of CoAP can be summarized as follows.

- A compact binary header (10–20 bytes), along with UDP, reduces the communication overhead, thereby reducing the delay and minimizing the power consumption due to data transmission.
- Since asynchronous data push is supported, it enables things to send information only when there is a change of observed state.
- This allows the things to sleep most of the time and conserve power.
- The minimal subset of REST requests supported by CoAP allows the protocol implementations to be less complex when compared to HTTP. This lowers the hardware requirements for the smart things on which it executes.

CoAP (CONTINUED)

- The M2M resource discovery is supported by CoAP to find a matching resource based on the CoRE link format.
- The draft CoAP proposal includes support for alternative non-IP messaging, such as Short Message Service (SMS) and transportation of CoAP messages over Bluetooth, ZigBee, Z-Wave, and so forth.

MQ Telemetry Transport (MQTT) protocol is another communication protocol designed for M2M communication, based on TCP/IP. Both CoAP and MQTT are expected to be widely used in IoT communication infrastructure in the future.

COORDINATION LANGUAGES

- A complete programming model can be built by combining two orthogonal models—a computation model and a coordination model.
- The computation model provides the computational infrastructure and programmers can build computational activity using them, whereas the coordination model provides the support for binding all those computational activities together.
- They argue that a computational model supported by languages such as C, by themselves cannot provide genuine coordination support among various computing activities.

COORDINATION LANGUAGES (CONTINUED)

- This observation is more relevant in IoT–Cloud programming, wherein there are numerous distributed activities which have to be coordinated in a reliable and fault-tolerant manner.

Coordination can be seen through two different perspectives:

(1) based on centralized control, named as Orchestration and

(2) based on distributed transparent control, named as Choreography.

- The W3C's Web services choreography working group defines Choreography as "the definition of the sequences and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state."

COORDINATION LANGUAGES (CONTINUED)

- Orchestration is seen as “the definition of sequence and conditions in which one single agent invokes other agents in order to realize some useful function.”
- There are many languages that provide Choreography and Orchestration support.
- Some of the features in coordination languages such as Linda, eLinda, Orc, and Jolie.

Linda and eLinda

- Linda is a coordination-programming model for writing parallel and distributed applications.
- It takes the responsibility of enforcing communication and coordination, while general-purpose languages such as C, C++, and Java are used for computational requirements of the application.
- The Linda model supports a shared-memory store called **tuple space** for communication between processes of the application.
- Tuple spaces can be accessed by simple operations such as “out” and “in.”
- These operations can be either blocking or nonblocking.
- CppLINDA is a C++ implementation of the Linda coordination model.

Linda and eLinda (CONTINUED)

- The eLinda model extends Linda. It adds a new output operation "wr" that can be used with the "rd" input operation to support broadcast communication.
- In Linda, if a minimum value of a dataset stored in a tuple space is required, all matching field values should be read, the reduction should be performed, and then the remaining data should be returned to the tuple space.
- While this procedure is accessing the tuple space to extract the minimum value, the tuple space is not accessible to other processes, which restricts the degree of parallelism by a large amount.

Linda and eLinda (CONTINUED)

- eLinda proposes the "**Programmable Matching Engine**" (PME) to solve problems such as the previous one.
- The PME allows the programmer to specify a custom matcher that can be used internally to retrieve tuples from the shared store.
- The PME has been found to be advantageous for parsing graphical languages and video-on-demand systems.

Orc

- Orc is a coordination language for distributed and concurrent programming.
- It is based on process calculus.
- It provides uniform access to computational services, including distributed communication and data manipulation.

A brief overview of the language features is as follows.

- The basic unit of computation in Orc is called a **site**, similar to a function or a procedure in other languages. The sites can be remote and unreliable.
- Sites can be called in the form of C(p); C is a site name and p is the list of parameters. The execution of a site-call invokes the service associated with the site. The call publishes the response if the site responds.

Orc (CONTINUED)

Orc has the following combinator-operators to support various compositions and work-flow patterns :

- Parallel combinator "|" is used for parallel, independent invocation. For example, in I | J, expressions I and J are initiated at the same time independently. The sites called by I and J are the ones called by I | J, and any value published by either I or J is published by I | J. There is no direct interaction or communication between these two computations.
- Sequential combinator "»" is used for invocations of sites in a sequential manner. In I>y>J, expression I is evaluated. Each value published by I initiates a separate and new execution of J. Now, the execution of I continues in parallel with the executions of J. If I does not publish even a single value, then there is no execution of J.

Orc (CONTINUED)

- Pruning combinator "«" is a special type of combinator which can be seen as an asynchronous parallel combinator. For example, in I<y<J, both I and J execute in parallel. Execution of parts of I which do not depend on y can proceed, but site-calls in I for which y is a parameter are suspended until y has a value. If J publishes a value which can be assigned to y, then J's execution is terminated and the suspended parts of I can then proceed.
- The "»" combinator has the highest precedence, followed by "|" and "«."

Orc (CONTINUED)

- Orc provides several fundamental sites, such as Rwait(t), Prompt(), and so forth, to promote writing efficient programs.

Orc allows users to define local functions. Function-calls act and look a lot like site-calls, with a few exceptions:

- A site call will block if some of its arguments are not available, but a function call does not.
- A site call can publish at most one value, but a function call can publish more than one value.

Orc also supports functions and sites as arguments to a function call.

The recent Orc implementation is allowing Java classes to be used as sites.

Jolie

- Jolie (Java Orchestration Language Interpreter Engine) is an orchestration language for services in Java-based environments.
- The statement composers and dynamic fault handling are two important features in this language.
- In dynamic fault handling, instead of fault handlers being statically programmed, they are installed dynamically at the execution time.
- This facilitates fine-tuning of fault handlers and termination handlers, depending upon which part of the code has already been executed.

Jolie (CONTINUED)

- In Jolie there are basically three statement composers: sequence, parallel, and input choice.
- Statements can be composed sequentially, using “;” operator.
- It means that the statement to the left of the sequence operator is executed first, and then the statement to the right of it.
- The syntax of the sequence statement is as follows.

```
statementx ; statementy
```

Jolie (CONTINUED)

- *statementx* gets executed first and then the *statementy*.
- The "|" operator is used to compose statements in parallel.
- The statements to the left and right of the parallel operator are executed concurrently.
- The syntax is as follows.

```
statementx | statementy
```

Jolie (CONTINUED)

- statementx and statementy are executed concurrently.
- The third composer is for guarded input.
- Here, message receiving is supported for any of the input statements that are listed.
- When a message for an input statement is received, all the other branches are deactivated and the corresponding branch behavior is executed.
- The syntax is as shown in the listing.

```
[IS_1]{branch_code_1}
[IS_2]{branch_code_2}
[IS_3]{branch_code_3}
```

Jolie (CONTINUED)

- If the message is received on the input statement IS_2, then branch_code_1 and branch_code_3 are disabled, and execution continues through branch_code_2.
- Since IoT is characterized by distributed execution, we believe explicit coordination- language support with at least minimal features for coordination and composition, for different work-flow patterns is a must for any IoT programming framework.

POLYGLOT PROGRAMMING

- Polyglot programming is also called multilingual programming.
- It is an art of developing simpler solutions by combining the best possible solutions using different programming languages and paradigms.
- This is based on the observation that there is no single programming paradigm or a programming language which can suit all the facets of modern-day programming or software requirements.
- It is also called poly-paradigm programming (PPP), to appreciate the fact that many modern-day software combines a subset of imperative, functional, logical, object-oriented, concurrent, parallel, and reactive programming paradigms.

POLYGLOT PROGRAMMING (CONTINUED)

- One of the oldest examples of polyglot programming is Emacs, which is a combination of parts written in both C and eLisp (dialect of Lisp).
- Web applications are generally based on three-tier architecture to promote loose coupling and modularity, and they are also a representation of polyglot software systems.
- Polyglot programming has been observed to have increased programmer productivity and software maintainability in web development.

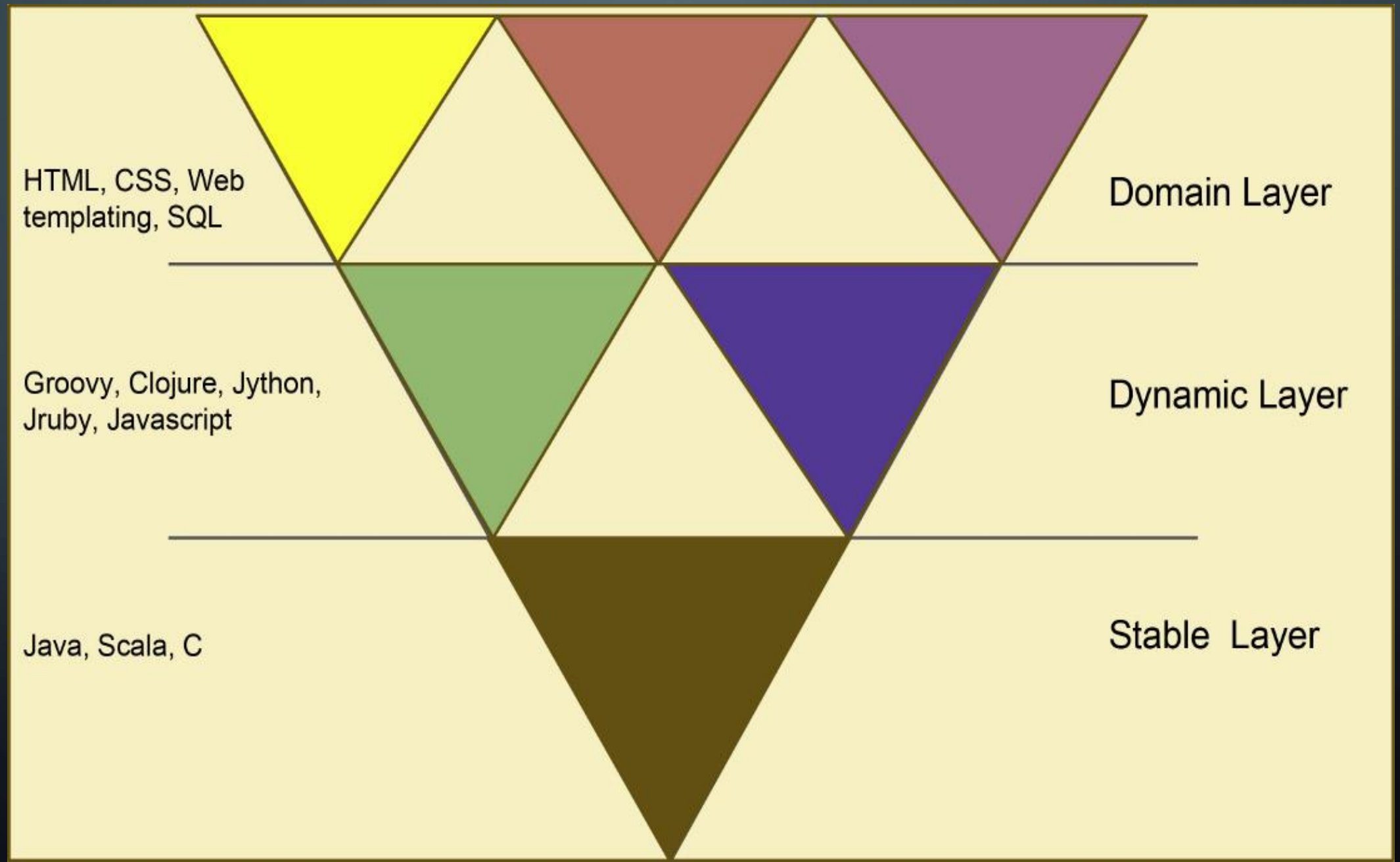
POLYGLOT PROGRAMMING (CONTINUED)

- One of the definitions says “programming in more than one language within the same context.”
- Another one says “using multiple programming languages on the same managed run-time.”
- Taking into account the developers’ perspective, the definition as: “programming in more than one language within the same context, where the context is either within one team, or several teams where the integration between the resulting applications require knowledge of the languages involved.”

POLYGLOT PROGRAMMING (CONTINUED)

- In a Polyglot programming environment, the platform used for the integration, and the different programming languages supported by the given platform are the two essential aspects.
- An Inverse pyramid can be used to categorize the programming languages in a polyglot software system.
- The Inverse pyramid has three layers: stable, dynamic, and domain, as shown in diagram below:

INVERSE PYRAMID FOR POLYGLOT PROGRAMMING



POLYGLOT PROGRAMMING (CONTINUED)

- Statically typed programming-languages such as Java and C that provide well-tested and stable functionality settle toward the stable layer.
- The less powerful general-purpose technologies, such as HTML and CSS, which are tightly coupled to a specific part of the application, bubble up to the top layer, and the dynamic layer in the middle consists of a variety of programming languages such as Groovy and Clojure, which are more flexible and aid rapid functionality development.
- The inverse pyramid signifies the fact that it is the single stable language, which supports all of the previously described layers and various languages in a bedrock fashion.

POLYGLOT PROGRAMMING (CONTINUED)

- Since IoT is characterized by heterogeneity in various forms, a single programming language or a single programming model may not be able to provide complete support for the application development in IoT.
- As we have already argued, at least a coordination language and a computational language is required in a unified programming model for IoT, which, in a way, is polyglot programming.

SURVEY OF IOT PROGRAMMING FRAMEWORKS

- These programming frameworks promote design reuse, implementation reuse, and validation reuse, thereby enhancing software extensibility, flexibility, and portability.
- The complexity of the domain and maturity of the problem are the biggest challenges in developing frameworks.
- Since IoT domain itself is in its initial stages, many frameworks are also in the development and experimental stages.

SURVEY OF IOT PROGRAMMING FRAMEWORKS (CONTINUED)

Following set of minimal features to be fulfilled by the programming frameworks for IoT:

- Coordination
- Heterogeneity
- Scalability
- Fault tolerance
- Lightweight footprint
- Support for latency-sensitive applications

SURVEY OF IOT PROGRAMMING FRAMEWORKS (CONTINUED)

Coordination -

- An IoT can have computing elements playing different roles: controllers, storage managers, and application processors.
- We need programming-language support for orchestrating their activities.
- The orchestration can be explicit (control driven) or implicit (data driven).

SURVEY OF IOT PROGRAMMING FRAMEWORKS (CONTINUED)

Heterogeneity -

- The IoT brings disparate computing devices together for the purposes of running smart-computing applications.
- The programming framework should be capable of efficiently exploiting the system heterogeneity by allowing the developer to provide guidance on how the computations must be mapped to the computing elements.

SURVEY OF IOT PROGRAMMING FRAMEWORKS (CONTINUED)

Scalability -

- For IoT to become a success, it is not sufficient to just interconnect massive numbers of devices.
- They should be programmed to run many creative applications, such that massive numbers of users would benefit from their deployment.
- Therefore, IoT needs programming frameworks that support a variety of programming patterns, and should also be able to perform load-balancing dynamically.

SURVEY OF IOT PROGRAMMING FRAMEWORKS (CONTINUED)

Fault tolerance -

- In IoT, we can expect frequent system partitioning due to the mobility of computing elements.
- The programming framework should allow developers to create applications that can gracefully go between online and offline states as networks partition and heal their connections.

SURVEY OF IOT PROGRAMMING FRAMEWORKS (CONTINUED)

Lightweight footprint -

- The programming framework should be lightweight both in terms of the runtime overhead and the programming effort needed by the developers.

SURVEY OF IOT PROGRAMMING FRAMEWORKS (CONTINUED)

Support for latency-sensitive applications -

- IoT will have many applications which would be geographically distributed and hence may be latency-sensitive.
- Pushing all the computations to Cloud will not help these sorts of applications.
- The programming framework, including the runtime, has to support these sorts of requirements dynamically.

IoT PROGRAMMING APPROACHES

The following four approaches are used predominantly in IoT application development

- Node-Centric Programming
- Database approach
- Macro Programming
- Model-Driven Development

IoT PROGRAMMING APPROACHES (CONTINUED)

Node-Centric Programming -

- Here, every aspect of application development, communication between nodes, collection and analysis of sensor data, and issuing of commands to actuator nodes has to be programmed by the application developer.
- Although it has greater control over the way that programs work, it is too labor-intensive and does not promote portability.

IoT PROGRAMMING APPROACHES (CONTINUED)

Database approach –

- In this model, every node is considered to be a part of a virtual database.
- Queries as part of an application can be issued on sensor nodes by the developer.
- This model does not support application logic at this level, rendering it to be of little use in IoT application development.

IoT PROGRAMMING APPROACHES (CONTINUED)

Macro Programming -

- In this methodology, application logic can be specified and also abstractions are provided to specify high-level communication, thereby hiding low-level details from developers, which will aid in modular and rapid development of applications.

IoT PROGRAMMING APPROACHES (CONTINUED)

Model-Driven Development:

- This takes note of both vertical and horizontal separation of concerns.
- Vertical separation increases the level of abstraction, thereby reducing application development complexity.
- Horizontal separation of concern reduces development complexity by describing the system, using different system views.
- Each perspective elaborates on a certain aspect of the system.

IoT PROGRAMMING APPROACHES (CONTINUED)

- Many of the IoT development kits, which are available in the market, support one of the previously listed approaches.
- This categorization is not exhaustive, as new hybrid approaches may evolve as the IoT domain itself matures.

EXISTING IOT FRAMEWORKS

- Mobile Fog
- ELIoT (Erlang Language for IoT)
- Compose API
- Distributed Dataflow Support for IoT
- PyoT
- Dripcast
- Calvin
- Simurgh
- High-Level Application Development for the Internet of Things
- PatRICIA

1. Mobile Fog

- Cisco has proposed a new computing model called Fog computing.
- Here, generic application logic is executed on resources throughout the network, including routers and dedicated computing nodes.
- In contrast to the pure Cloud paradigm, fog-computing resources perform low-latency processing near the edge, while latency-tolerant, large-scope aggregations are performed on powerful resources in the core of the network (Cloud).

1. Mobile Fog (CONTINUED)

- Mobile fog extends fog computing by providing a Platform as a Service (PaaS) programming model for IoT application development to simplify the task of application development that runs on heterogeneous devices distributed over a wide area, and also to provide support for dynamic scaling based on their workload.

1. Mobile Fog (CONTINUED)

- Here, an application will contain processes distributed throughout the fog-computing infrastructure, on Cloud and on edge devices, based on geographical proximity and hierarchy.
- Each process can perform tasks with respect to its location and level in the network hierarchy, such as sensing, actuation, and aggregation.
- A process running on a device which is at the edge is a *leaf node*, whereas a process in the Cloud is the *root node* in a given hierarchy.
- Processes on nodes between devices and Cloud are intermediate nodes (routers, servers, etc.).
- Each process handles a workload from a certain geospatial region.

1. Mobile Fog (CONTINUED)

- Mobile Fog provides API support through its runtime. Mobile Fog uses computing- instance requirements to provide dynamic scaling.
- It is based on user-provided policy, such as CPU utilization rate, bandwidth, and so forth.

2. ELIoT (Erlang Language for IoT)

- Although the language Erlang was originally designed for embedded platforms, over a period of time it amassed a complex infrastructure, which is usually not required in devices and is a burden on resource constrained things.
- ELIoT, Erlang language for IoT, tries to address this for IoT application development.

2. ELIoT (Erlang Language for IoT) (CONTINUED)

- ELIoT provides a small library for developing decentralized sensing/actuation systems, an interpreter suited for resource-constrained IoT devices, and a simulator for testing the implementations in a fully or partially simulated environment.
- The ELIoT's virtual machine is a stripped-down, lightweight version of Erlang's virtual machine. Heavy libraries, which are not required for IoT, are removed (such as CORBA middleware systems).
- It includes a custom-networking stack for improving efficiency and for supporting new communication primitives.
- Instead of TCP, UDP is used for both reliable and nonreliable communication. A customized reliability layer is built on UDP.

2. ELIoT (Erlang Language for IoT) (CONTINUED)

- Generally for IoT applications, strict layering of the networking stack may not be fully advantageous; some form of cross-layering is found to be helpful for IoT applications.
- Erlang's network driver fills the incoming-message queue of the receiver with the payload of the message, hiding all of the other details, whereas the network driver of the ELIoT exposes additional information such as the IP address of the source node and the Received Signal Strength Indicator (RSSI) coming from the radio, which are treated as any other type of data.

2. ELIoT (Erlang Language for IoT) (CONTINUED)

- The Erlang's uni-cast interprocess communication operator "!" is built on a complete TCP/IP stack, ensuring reliable communication for both local and remote communication.
- Since, TCP/IP stack comes with a cost and can be resource-draining on devices, in ELIoT, the ! operator is used only for communication between local processes.
- Remote communications in ELIoT are handled by a set of specific functions from the ELIoT library, whose semantics are its best effort, and is limited to SingleHop in wireless networks.
- Furthermore, the ELIoT library supports a rich set of communication patterns, including the broadcast mode.

2. ELIoT (Erlang Language for IoT) (CONTINUED)

- ELIoT provides a simulator for supporting IoT application debugging and testing.
- The simulator can model a complete system through virtual nodes running unmodified ELIoT code.
- Also, it can run a mixed deployment where virtual nodes seamlessly interact with physical devices.
- The ELIoT simulator allows for debugging a system in a fully simulated deployment environment, which can seamlessly move into an actual deployment environment.

2. ELIoT (Erlang Language for IoT) (CONTINUED)

- The ELIoT framework provides wrappers on nodes, which are basically RESTful interfaces, through which nodes can be accessed by users through the normal HTTP operation.
- ELIoT brings in the advantages of Erlang to IoT in a lightweight framework.

3. Compose API

- Compose API is an IoT service-provider platform through RESTful APIs, wherein things, users, and the Compose platform can interact with each other to provide services based on IoT, called Internet of Services (IoS).
- Compose platform is based on Web of Things (WoT): all of the physical objects connected to the platform are web-enabled and can interact among themselves using the web protocols.
- Along with the APIs, the compose platform consists of GUI, semantic registry, cloud runtime, and communication libraries.

3. Compose API (CONTINUED)

- Any object which implements the communication protocols of the Compose API is web- enabled and is called a Web Object (WO).
- Each WO holds a virtual identity inside the Compose platform, called the Service Object (SO).
- The SOs communicate with the external WOs through APIs.
- SOs can act as data endpoints or they can also act as intermediaries, feeding processed data to other SOs.
- Every time a sensor attached to WOs produces a new reading, it is forwarded as a Sensor Update (SU) on a stream to the Compose platform, to be collected by the corresponding SO for processing based on some processing logic.

3. Compose API (CONTINUED)

- The processing logic is a combination of logical, string, and arithmetic operations implemented in the form of a processing pipeline.
- An SU goes through a number of stages in the pipeline in order to transform into a new output or a new SU.
- Connections between SOs are built through subscriptions, and communication between them is through events.
- A JSON document description deploys each SO in the platform.

3. Compose API (CONTINUED)

- Compose API simplifies node-centric programming and exposes nodes through RESTful APIs which can be further composed.
- Such a programming methodology is quite advantageous to IoT.

4. Distributed Dataflow Support for IoT

- In this approach, existing IoT dataflow platforms such as WOTkit processor and Node-RED are extended to support distributed dataflow, which is one of the important characteristic features of IoT.
- Dataflow programs are generally called flows, which consist of nodes connected by “wires.”
- The dataflows are generated using JSON documents.
- During execution, nodes get instantiated in the memory, and the code is executed as and when the node receives data on the incoming “wire.”

4. Distributed Dataflow Support for IoT (CONTINUED)

- The nodes do not share states with each other and are inherently independent and can execute code in parallel.
- This facilitates computation migration between heavy processors and devices seamlessly.
- Based on user choices and trade-offs, computations can be split and distributed, so that a part of them can execute in the cloud while the other parts can execute on edge devices.

4. Distributed Dataflow Support for IoT (CONTINUED)

- According to the authors, the present-day IoT dataflow platforms need to be extended to support distributed dataflow, for which three things are necessary: flow ownership, naming of nodes, and classification of connections (wires) as local or remote.
- This framework aims to incorporate these three attributes to WOTkit and Node- RED to aid in IoT application development from the dataflow perspective.

5. PyoT

- PyoT is a programming framework for WSNs, which have the capability to communicate with each other through the Internet, using 6LoWPAN and CoAP.
- PyoT abstracts WSNs as software objects, which can be manipulated and composed to perform complex tasks.
- PyoT uses CoAP's RESTful interface to interact with nodes.
- Applications can consider sensing and actuating capabilities of nodes, shared with the external world through URIs.
- The users can discover available resources, monitor sensors and actuators, store data, define events and actions, and program to interact with resources using Python.
- PyoT supports "in-network processing," in which a part of the application logic can be directly run on devices.

5. PyoT (CONTINUED)

- PyoT has five components: (1) Virtual Control Room, (2) Shell, (3) Storage Element, (4) Message Queue, and (5) one or more PyoT Worker Nodes.
- The Web user interface is the virtual control room that allows execution of basic operations, such as listing of resources, sensor monitoring, and data storage.
- The Shell allows macro programming for defining complex operations through a set of Python APIs for interacting with resources, which are abstracted as Python objects.

5. PyoT (CONTINUED)

- The Storage Element maintains the system status. Each PyoT Worker Node generally manages an IoT-based WSN, by providing a set of processes that perform generic tasks and support communication activities with other nodes.
- The PyoT Worker Node also keeps track of nodes and their resources, provides updates to the Storage Element, performs sensor-data collection, and also supports event detection.
- The macro programming support by this framework will lessen the burden on IoT programmers.

6. Dripcast

- Dripcast is a Java-based application-development framework to integrate smart devices into the cloud-computing infrastructure.
- Further, it is a serverless framework for storing and processing Java objects in a cloud environment.
- These Java objects will be made available on smart things, and users can manipulate those objects as if they were local objects.

6. Dripcast (CONTINUED)

- It implements transparent Java remote procedure- calls and a mechanism to read, store, and process Java objects in a distributed, scalable data-store.
- Under Dripcast, all Java objects have a worldwide unique ID.
- The Dripcast framework consists of four components: Client, Relay, Engine, and Store.

6. Dripcast (CONTINUED)

1. Client is a Java library, which works on devices such as smartphones and tablets. It monitors the Java object on the client devices, and forwards remote procedure- calls, which are abstracted from the users to the Relay.

2. Relay is a stateless distribution-gateway. It forwards requests from clients to corresponding engine servers. A relay server knows the association of the object's unique ID and engine servers; a Distributed Hash Table (DHT) manages this association.

6. Dripcast (CONTINUED)

3. Engine is a set of engine servers. Each engine server runs JavaVM and executes Java methods of an object for a remote procedure-call request forwarded by the relay, and returns the result back to the relay. If there is a state change of the object, then the new state is stored back into the Store.

4. Store is a scalable distributed data-store for storing Java objects with the capability for replication and automatic recovery.

- The Dripcast framework enables Java-based IoT application development.

7. Calvin

- It is a framework that merges IoT and cloud in a unified programming model.
- It combines the ideas of actor-model and flow-based computing.
- To simplify application development, it proposes four phases to be followed in a sequential fashion: Describe, Connect, Deploy, and Manage.

7. Calvin (CONTINUED)

1. Describe: In this phase, the functional parts of the applications, which are reusable components, are described. In Calvin, everything is treated as an actor: devices, services, and even a piece of computation on cloud. These actors can communicate with each other through ports. To create an actor, a developer describes the actions, their input/output relations, the conditions for a particular action to be triggered, and also the priority between actions. Device manufacturers can supply actors that correspond to their devices as part of the support code shipped with their devices, thus enabling their devices to easily integrate with a Calvin application.

7. Calvin (CONTINUED)

2. Connect: Once the actors have been described, the next step is to connect those actors by directed graphs between the ports of a number of actors.

7. Calvin (CONTINUED)

- 3. Deploy: In this phase, an application is instantiated according to the graphs provided with its description. The description/connect phase does not specify where the various actors should execute, nor how the data should be transported between them. This is handled during deployment of the application. The distributed runtime present at the nodes where the application gets deployed shoulders this responsibility. By forming a mesh network of runtime on nodes, actors in a running application can migrate from one runtime to another. Once the runtime has been instantiated and connected to the actors locally, the distributed execution environment can move actors to any accessible runtime based on resources, locality, connectivity, and performance requirements.

7. Calvin (CONTINUED)

4. Manage: In this phase, the distributed execution environment monitors the applications, handling migration of actors, updates, error recovery and scaling, along with book-keeping.

7. Calvin (CONTINUED)

- These phases are supported by the runtime, APIs, and communication protocols.
- The platform dependent part of Calvin runtime manages communication between runtimes, transport layer support, the inter-runtime communication, and abstraction for I/O, sensing mechanisms to the upper levels of the runtime.
- The platform-independent runtime provides interface for the actors.
- The scheduler of the Calvin runtime resides in this layer.
- Calvin runtime supports multitenancy.
- Once an application is deployed, actors may share runtime with actors from other applications.

8. Simurgh

- Simurgh provides a high-level programming framework for IoT application development.
- The framework supports the exposing of IoT services as RESTful APIs, and also the composing of those IoT services to create various flow patterns in a simplified manner.
- The overall Simurgh architecture has two main layers: **Thing layer and Platform layer**.
- In the **thing layer** there is a software component called **Network Discovery and Registration Broker**, which listens to the incoming connection requests from devices and handles them.

8. Simurgh (CONTINUED)

- There is a rich set of libraries providing device-specific interfaces.
- An API mediator assists programmers to expose their applications through RESTful APIs.
- Also, they provide RESTful wrappers for those low-level device interfaces which are not supported by native vendors, and, finally, an API manager which monitors API's access from the external world.

8. Simurgh (CONTINUED)

The platform layer has the following components:

- Thing Description Repository
- Two-Phase Discovery Engine
- Flow Design
- Flow Composition
- Flow Execution Engine
- Flow Template Management and Repository
- Request Management

8. Simurgh (CONTINUED)

1. Thing Description Repository: This stores information about things and services offered by them, periodically updated by the Network Discovery and Registration Broker and API Mediator component. Things are described using TDD (Thing Description Document). A TDD file consists of mainly two parts:

a. Entity Properties: Usually a user-chosen name, last modification date and entity's location is stored.

b. Entity Services: For each of the entities described earlier, entity services define APIs that are available on the entity. These API definition files can be in RESTful API Modeling Language (RAML) or in Swagger format.

8. Simurgh (CONTINUED)

2. Two-Phase Discovery Engine: This is used to discover an entity and its corresponding APIs in two phase. In the first phase, the engine will search in TDD repository to find entities based on given criteria. If the goal is finding an API of an entity capable of doing a certain task, then, another search is performed on their respective API Description Documents.

8. Simurgh (CONTINUED)

3. Flow Design: This component assists in designing flows, which are chains of IoT services. Through this component, users can discover things, discover their APIs, and also can call the found APIs, thereby generating a flow.

8. Simurgh (CONTINUED)

4. Flow Composition: Two or more flows can be combined to build a new flow that can deliver a new functionality. This component performs those compositions.

8. Simurgh (CONTINUED)

5. Flow Execution Engine: This engine provides all the required resources during the execution of a flow. It configures them and executes all the necessary APIs to fulfill a request.

8. Simurgh (CONTINUED)

6. Flow Template Management and Repository: Flows are managed, and, also, to promote flow reusability, the used patterns are stored and are exposed to users when they are designing new flows.

8. Simurgh (CONTINUED)

7. Request Management: This component performs user-request matching to flow templates. If a match is not found, then the request will be forwarded to the Flow Composition module to match with composed flow patterns. Furthermore, if a flow is not found, then users can build the required flow using a flow-design interface.

The Simurgh framework provides detailed support for IoT development. Assistance to develop, manage, and reuse flow patterns as provided in this framework is crucial for IoT programmers.

9. High-Level Application Development for the Internet of Things

- The authors propose a detailed framework for developing IoT applications.
- They propose a new developmental methodology and a framework to support it.
- To simplify the process of IoT application development, this framework stresses identifying stakeholders and demarcating (separate) their responsibilities.
- They can be domain experts, software designers, application developers, device developers, and network managers.

9. High-Level Application Development for the Internet of Things (CONTINUED)

- In this framework, a conceptual model, which serves as a knowledge base for a problem, is built, taking into account four different areas of concern in IoT application development: domain-specific concepts, functional-specific concepts, deployment-specific concepts, and platform-specific concepts.

9. High-Level Application Development for the Internet of Things (CONTINUED)

- Domain-Specific Concepts: The concepts in this category are unique to an application domain. For example, building automation is reasoned in terms of rooms and floors. There can be sensors, actuators, and storage devices too. These concepts are identified under: Entity of Interest (EoI), which can be any object (eg, room, book, plant); Resources, such as sensors, actuators, and storage devices; and the Region used to specify the location of a device.

9. High-Level Application Development for the Internet of Things (CONTINUED)

- Functionality-Specific Concepts: These concepts describe computational elements of an application and interactions among them. These computational elements are software components that encapsulate and hide a subset of the system's functionality and data. Interactions among software components happen through request/response, publish/subscribe, and command mode.

9. High-Level Application Development for the Internet of Things (CONTINUED)

- Deployment-Specific Concepts: These concepts describe information about devices. A device is an entity that provides resources the ability to interact with each other. Each device can host zero or more resources and is located in a region.
- Platform-Specific Concepts: These are computer programs that act as a translator between hardware devices and an application. They are categorized as Sensor driver, Actuator driver, Storage Services, and End-user application.

9. High-Level Application Development for the Internet of Things (CONTINUED)

The developmental framework consists of modeling languages and automation techniques to support stakeholders to implement the conceptual model.

- Support for Domain Concerns: The developmental framework supports domain concerns in specifying the domain vocabulary using Srijan Vocabulary Language and compiling those vocabulary specifications. This compiled output supports the later phases.
- Functional Concerns: For this phase the developmental framework supports by specifying application architecture using Srijan Architecture Language, then compiling the architecture specification, and finally by implementing the application logic

9. High-Level Application Development for the Internet of Things (CONTINUED)

- Deployment Concerns: The framework specifies the target deployment of devices using the Srijan Deployment Language, and maps a set of computational services to a set of devices.
- Platform Concerns: Here, the device drivers are implemented, and the linker generates packages that can be deployed on devices. It basically combines output of all the preceding phases, such as application logic and device drivers. Device-specific code is generated in this phase.

This framework takes the software-engineering approach to IoT development and supports model driven development.

10. PatRICIA

- PatRICIA is a programming framework for IoT application development on Cloud platforms.
- The key feature of this framework is the “intent”-based programming model.
- The programmers can specify the intent and the scope of the intent.
- Intents can be either a monitoring task on devices or a controlling task of devices.
- The intent scope delimits the range of an intent.

10. PATRICIA (CONTINUED)

- It is the responsibility of the framework to execute the intent on the devices demarcated by the scope of the intent.
- This programming model hides many of the underlying complexities of IoT programming from the end users.
- For example, if PatRICIA is being used in traffic management, then the end user can simply say “track all the vehicles exceeding the speed limit 90.”
- Here, PatRICIA executes the intent: track the vehicles; and the scope of the intent: all those vehicles, which exceed the speed limit 90 kmph.

10. PatRICIA (CONTINUED)

- The architecture of the framework is **four-tiered**.
- The **topmost** layer is named **Development Support Layer**.
- It contains tools to aid in the application-development lifecycle.
- It has a module called **Application Manager**, whose responsibility is to configure, deploy, and license applications, along with providing a testing environment for IoT applications.

10. PATRICIA (CONTINUED)

- The important part of this layer is that it exposes the programming model based upon “intent” to developers.
- The **Cloud System Runtime layer** provides support for intent-based programming by executing the intent on the “scope of the intent.”
- The **Data and Device Integration layer** is responsible for data Management, IoT- device management, and virtualization.
- The **Device Communication layer** implements different connectors catering to heterogeneous devices. The physical layer has all the things, which can communicate through the Internet.

10. PatRICIA (CONTINUED)

- Intent-based programming model: This programming model provides tools to work with monitoring and control tasks.
- Control tasks help developers to operate, provision, and manage low-level components.
- They provide a high-level representation of underlying devices and their functionality.

10. PATRICIA (CONTINUED)

- They are named “ControlIntent.” Likewise, monitoring tasks, named “MonitorIntent,” are used to subscribe to events from the physical environment, along with obtaining and provisioning devices’ context.
- These tasks can be represented by application developers as “intents,” which get automatically instantiated for the supplied intent scope.

10. PatRICIA (CONTINUED)

- Intent is a data structure representing a specific task, which can be performed in a physical environment.
- Based on the specified intent, a suitable task is selected (control or monitor), instantiated, and executed on the Cloud platform.
- The Intent thereby gets translated as a sequence of steps to process data or to perform some actuation on the underlying things.

10. PATRICIA (CONTINUED)

- To subscribe to an event in the underlying physical environment or to perform some IoT control, developers can define and configure intents.
- This shields the developers from the inherent complexity of the IoT.

10. PatRICIA (CONTINUED)

- Intent scope is an abstraction of a group of physical entities which have some common properties.
- The demarcation of the physical layer for an intent scope is determined on the Cloud.
- By specifying the properties that have to be satisfied by physical entities to be in a scope, developers define Intent Scope.

10. PATRICIA (CONTINUED)

- PatRICIA also provides operators such as send, notify, poll, and delimit to work with intents.
- The support for intent-based programming in PatRICIA will hide many of the underlying heterogeneity, which is advantageous in IoT programming.

Table 5.1 Highlights of Various IoT Programming Frameworks

Framework	Approach	Key Features	Program's Target	Coordination Support
Mobile Fog	Macroprogramming	Edge processing, dynamic scaling, cloud support, Runtime API support.	Devices	Coordination support through special APIs.
ELIoT	Macroprogramming	Extends Erlang for IoT, Support for broadcast communication, RESTful API support, simulator, virtual machine support.	Devices	Coordination support through Erlang language.
Compose API	Macroprogramming	RESTful APIs to access things, cloud support, composing of services through APIs.	Cloud and devices	Coordination support on Cloud.
Distributed Dataflow support for IoT	Macroprogramming, with dataflow support	Dataflow-based IoT application development, edge processing.	Devices	Coordination through choreography.
PyoT	Macroprogramming	Edge processing support for latency-sensitive applications, Python for macroprogramming, URIs for nodes, RESTful APIs.	Devices and web	Coordination support through macroprogramming.
Dripcast	Model-driven development (Java)	Services in terms of Java objects, remote management of objects, Cloud support.	Devices	No explicit coordination support.
Calvin	Model-driven development	Actor model and dataflow- based development, Cloud support, runtime multitenancy support for things.	Devices	Coordination through choreography.
Simurgh	Macroprogramming	RESTful API support, flow design and composition support with reusability.	Devices	Orchestration support for flow patterns.
High-level application development for IoT	Model-driven development (own languages)	Complete application development, lifecycle support, division of responsibilities between stakeholders, new languages for vocabulary, architecture, and deployment specification.	Devices	Coordination support specified during identification of functional concerns.
PatRICIA	Model-driven development	Intent-based programming, cloud support for control and monitoring of tasks.	Devices and Cloud	Coordination support on cloud, specified through scope of the intent.