A Study on Software Engineering Design Patterns

Advanced Topic Review Final Report

Advanced Software Engineering

Varaprasad Rao Kurra (email: vkurra1@student.gsu.edu)

Sravanthi Malepati (email: smalepati1@student.gsu.edu)

Abstract: In Software Engineering, Design patterns are some of the best practices adopted by software developers while writing source code for new application systems. Some of the most common problems during the software design stage are redundancy, complexity, and Debugging. These design patterns can describe the issues, provide a relevant solution, and recommend the appropriate time to apply it. It is important to use design patterns because they offer an established solution; if so, followed solves the recurring problem. Our review focuses on what design patterns must be used in leading-edge technologies like the Machine learning [1], Internet of Things (IoT), and Deep Learning. For example, Machine Learning applications have a pipe and filter design [2] pattern while detecting a cancer cell, whereas Deep Learning has a multi-layered design pattern approach for intrusion detection. Secured design patterns were developed for IoT software system [3]. We do a systematic study to collect, classify, and discuss these software engineering design patterns in cutting-edge technologies. Our review collects software engineering design patterns for IoT, Machine Learning, and Deep Learning techniques to provide developers with a comprehensive classification of such patterns.

***Keywords* — Design Patterns, Machine Learning Design Patterns, IoT Design Patterns, Machine Learning Algorithm, Security in IoT Systems.**

I. Introduction

Every software engineering student is aware of the traditional design patterns that have been existing and making the software design easy for the software developer. These established design patterns provide a solution to the recurring problem in the implementation and designing the software. These design patterns are organized into Creational, Structural and Behavioral design patterns depending upon how and when these patterns are used in the object-oriented design of the software. Mostly design these design patterns are used as a part of the object-oriented level. As computer science students we also need to know the design patterns that is being researched and reported in the field of computer science. We will introduce the design patterns reported in the Machine Learning for the pre-processing, training a model and selecting a model. These design patterns will be very useful for a data scientist and a machine learning developer. We will also introduce the design patterns in the Internet of Things design patterns at the architectural level how IoT developers use a reported design pattern to solve the recurring problem. In the first stage of the report, we will introduce the design patterns in Machine Learning, followed by IoT design patterns.

II. Design Patterns in Machine Learning

A. *Why design patterns in Machine Learning*

The process of building a Machine Learning model presents a variety of unique challenges and difficulties that have an impact on Machine Learning design. Data Quality refers to data accuracy, data completeness, data consistency and timeliness. We have a set of design patterns reported to handle these in the machine learning model. Design patterns are a way to codify the knowledge and experience of experts into advice that all practitioners can follow. Design patterns in Machine Learning have been categorized into five types. Namely design patterns in Data Representation, Problem Representation design patterns, Model Training design patterns, design patterns for Resilient Serving and Reproducibility design patters. We will be going through each of these design patterns with an example and, then we will look into an example of Software Engineering design pattern example used in Machine Learning.

B. ***Data Representation Design Patterns***

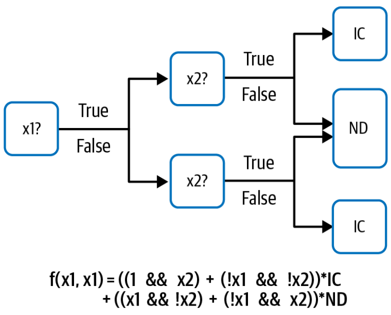
The heart of any machine learning model can be expressed as a mathematical function that is defined to operate on specific types of data. For example, linear regression is nothing but a mathematical equation that can fit into a straight-line equation y = mx+c. At the same time, real-world machine learning models need to operate on data that may not be directly pluggable into the mathematical function. Some of the simple data representation are Numerical Inputs, Linear Scaling, Clipping, Z score normalization, One Hot encoding. Let us discuss the first design pattern in Data Representation Hashed Feature.

Problem:

One-hot encoding a categorical input variable requires knowing the vocabulary beforehand. This is not a problem if the input variable is something like the language a book is written in or the day of the week that traffic level is being predicted. Below is the data representation in the Decision Tree.

Solution:

The Hashed Feature design pattern demonstrates a categorical input variable by performing the below-mentioned steps. First, we renovate the categorical input into a unique string. Secondly, we invoke a deterministic and portable, so that this algorithm will be utilized in both training and serving the hashing algorithm on the string. At last, we take the remainder when the hash result is divided by the desired number of buckets.


x1?
True
False
x2?
x2?
True
False
True
False
IC
ND
IC
f(x1, x1) = ((1 && x2) + (!x1 && !x2))*IC
+ ((x1 && !x2) + (!x1 && x2))*ND

Why does it work:

It's easy to see that the *high cardinality* [the frequency of the attribute] problem is addressed as long as we choose a small enough number of hash stacks. Even if an airport with a handful of observations is not part of the training dataset, its hashed feature value will be in the range [0–9]. The *cold-start* situation is similar to the out-of-vocabulary situation.

C. *Problem Representation Design Pattern*

Both input and output forms are two key factors impacting the model architecture. For instance, the output in supervised machine learning problems can vary depending on whether the problem being solved is a classification or regression problem. The *Ensemble* design pattern solves a problem by training multiple models and aggregating their responses.

Problem:

Suppose we have trained a weight calculation model, engineering special features and tallying additional layers to developed neural network model so that the error on our training set is nearly zero. Excellent, you say! However, when we examine to use our developed model in production server at the hospital or calculate performance on the hold out trial set, our expectations are all incorrect. What happened? And more suggestively, how can we fix it?

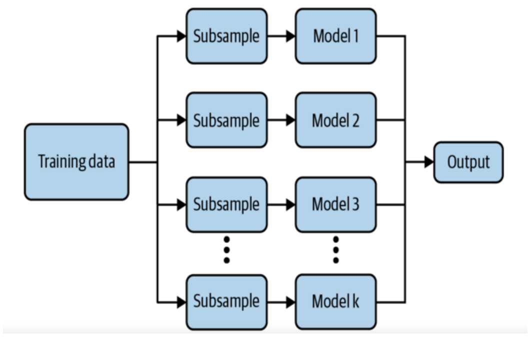
No ML model is faultless. To better understand where and how our model is mistaken, the error of an ML model can be shattered down into three parts: the *irreducible error*, the *error due to bias*, and the *error due to variance.*

Solution:

Ensemble methods are the meta procedures that chain several machine learning models as a method to lessening the bias and variance and improve model performance. In general, the idea is that combining multiple models helps to improve machine learning results. By constructing several models with distinct inductive biases and aggregating their outputs, we hope to get a model with better functioning. In this section, we'll discuss some commonly used ensemble methods like bagging, boosting, and stacking.

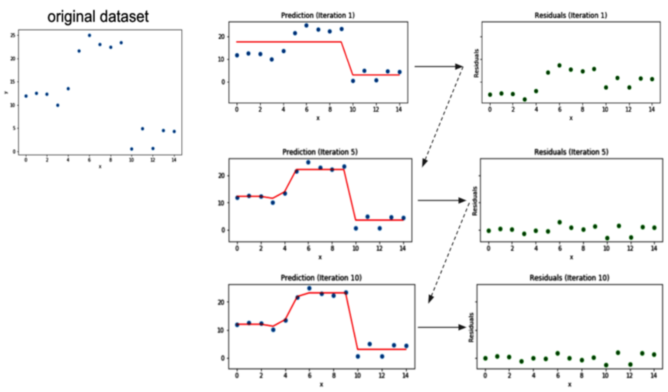
Bagging:

Bagging, i.e., a short step for bootstrap aggregating is a type of parallel ensembling method and is used to tackle high variance in machine learning algorithms generated models. The bootstrap part of bagging refers to the datasets used for training the ensemble members. Specifically, if there are *k* sub models, then there are *k* separate datasets used for training each sub-model of the ensemble.



Boosting:

Boosting is different Ensemble method. However, unlike bagging, boosting eventually constructs an ensemble model algorithm with more volume than the individual member models. For this reason, boosting provides a more efficient means of decreasing bias than inconsistency. The impression behind boosting is to iteratively build an ensemble of models where each consecutive model focuses on learning the examples the previous model got wrong.



Stacking:

Stacking is an ensemble method that combines the outputs of a collection of models to make a prediction. The initial models, which are typical of different model types, are trained to completion on the full training dataset. Then, a secondary meta-model is trained using the initial model outputs as features. This second meta-model learns how to best combine the outcomes of the initial models to decrease the training error and can be any type of machine learning model.

Why it works:

Boosting mechanism works well because the model is disciplined more and more conferring to the residuals at each iteration step. With each iteration, the ensemble model is heartened to get better and better at forecasting those hard-to-predict examples. Stacking works because it combines the best of both bagging and boosting. The secondary model can be thought of as a more high-level version of model averaging.

D. *Model Training Design Pattern*

Machine learning models are typically trained iteratively, and this iterative process is familiarly called the *training loop*. Here we will discuss what the normal training loop looks like and catalog a number of circumstances in which you might want to do something dissimilar.

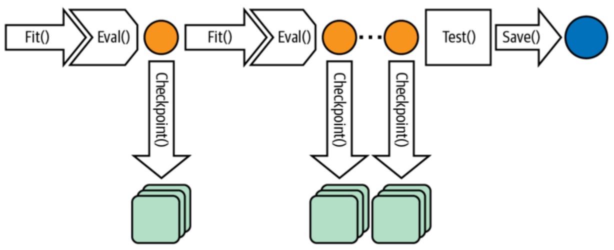
Problem:

The more complex a model is (for example, the more layers and nodes a neural network has), the larger the dataset that is needed to train it effectively. This is because more complex representations tend to have more tunable parameters. As model sizes increase, the time it takes to fit one batch of examples also increases. As the data size increases the number of batches also increases. Therefore, in terms of computational complexity, this double whammy means that training takes a extended time.

Solution:

At the end of every epoch, we can save the model state. Then, if the training loop is intermittent for any reason, we can go back to the saved model state and restart. Yet, when doing this, we have to make sure to save the middle model state, not just the model. What does that mean? Once training is complete, we save or *export* the model so that we can deploy it for inference. An exported model does not contain the entire model state, just the information necessary to create the prediction function.

Saving the full model state so that model training can restart from a point is called *checkpointing*, and the saved model files are called *checkpoints*. How often should we checkpoint? The model state variations after every batch because of gradient descent. So, if we don't want to lose any work, we should checkpoint after every batch. However, checkpoints are huge, and this I/O would add considerable overhead. Instead, model frameworks typically provide the option to checkpoint at the end of every epoch.



Checkpoints in PyTorch:

When loading from a checkpoint, you need to create the necessary classes and then load them from the checkpoint:

Save the model:

```
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            …
            }, PATH)
```

Loading the saved model from a specific Checkpoint:

```
model = ...
optimizer = ...
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

E. *Reproducability Design Pattern*

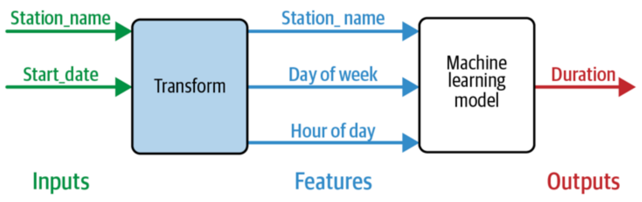
During training, machine learning models are prepared with arbitrary values and then adjusted based on training data available. K-means algorithm is implemented by *scikit-learn* which require setting the random_state in order to ensure the algorithm returns the same results each time:

```
def cluster_kmeans(X):
    from sklearn import cluster
    k_means = cluster.KMeans(n_clusters=10, random_state=10)
    labels = k_means.fit(X).labels_[::]
    return labels
```

Problem:

The problem is that the *inputs* to a machine learning model are not the *features* that the machine learning model uses in its computations. In a text classification model, for example, the inputs are the raw text documents, and the qualities are the numerical entrenching representations of this text. When we train a machine learning model, we train it with features that are obtained from the raw inputs.

Beyond the random seed, there are many other objects that need to be static in order to ensure reproducibility during training. In addition, machine learning contains of diverse stages, such as training, deployment, and retraining. It is often significant that some things are reproducible across these stages as well.



Solution:

The solution is to clearly capture the conversions applied to convert the model inputs into features. In BigQuery ML, this is implemented by using the TRANSFORM clause. Using TRANSFORM ensures that

these transformations are automatically applied during ML.PREDICT.
Given the provision for TRANSFORM, the model above should be rewritten as: Notice how we have clearly detached out the inputs (in the SELECT clause) from the features (in the TRANSFORM clause). Now, prediction is much easier. We can simply send to the model the station name and a timestamp (the inputs):

```
CREATE OR REPLACE MODEL ch09eu.bicycle_model
OPTIONS(input_label_cols=['duration'],
        model_type='linear_reg')
TRANSFORM(
 SELECT * EXCEPT(start_date)
 , CAST(EXTRACT(dayofweek from start_date) AS STRING)
 as dayofweek -- feature1
 , CAST(EXTRACT(hour from start_date) AS STRING)
 as hourofday –- feature2
)
AS
SELECT
 duration, start_station_name, start_date -- inputs
FROM
 `bigquery-public-data.london_bicycles.cycle_hire`
```

We can simply send to the model the station name and a timestamp (the inputs):

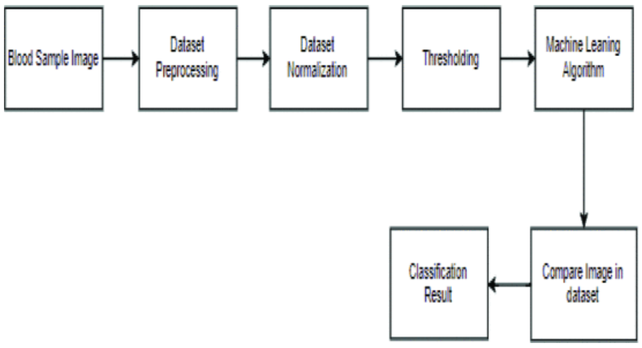
```
SELECT * FROM ML.PREDICT(MODEL ch09eu.bicycle_model,(
   'Kings Cross' AS start_station_name
 , CURRENT_TIMESTAMP() as start_date
))
```

If we are using a framework where support for the Transform design pattern is not built in, we should design our model architecture in such a way that the transformations carried out during training are easy to reproduce during serving.

Applications of Design Patterns in Machine Learning

Cancer Cell Detection using Machine Learning

Microscopic images are evaluated visually by hematologists and the procedure is monotonous and time taking which causes late uncovering. Therefore, automatic image handling context is required that can overcome related limitations in visual examination which provide early detection of disease and also type of cancer.



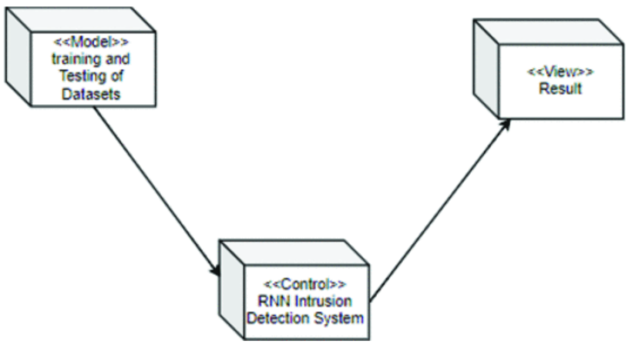
Design Pattern:

The system design was established especially to obey to the objective of having two sets of image sets, one for training and the other for testing.

A composite pattern would be suit for a model like this. The whole system is divided into components which lead to the result which classifies the blood cells as cancerous or otherwise.

Intrusion Detection using Deep Learning

Recurrent neural networks include input units, output units and hidden units. The model is a one-way flow of information from the input nodes to the hidden nodes. There is a combination of the one-way data flow from the previous time-based seclusion unit to the current timing hiding unit.


<<Model>> training and Testing of Datasets
<<View>> Result
<<Control>> RNN Intrusion Detection System

Design Pattern:

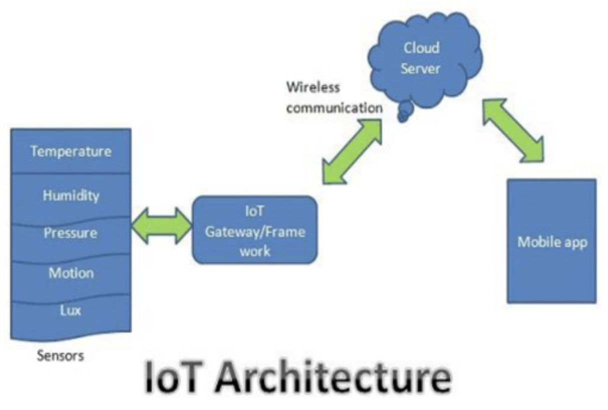
The training set, data pre-processing and its training play many roles when attached to other detection systems or systems that widely use machine learning techniques. The model is a unique and simple representation of how an efficient and quality software design can be presented.

III. Design Patterns in Internet of Things

A. IoT Architecture

The Internet of Things (IoT) can be defined as the concept of taking material things and then connecting these to the Internet. The IoT system consists of billions of physical devices, which collect and share data over the Internet. Structure of Internet of things is based on four simple building blocks, also called IoT architecture layers. As shown below, the figure IoT architecture layer consists of four simple blocks.

Sensors are everywhere; sensors will sniff a wide variety of information from Location, Weather/Environment conditions, running machine, from the human body, engine maintenance data to health essentials of a vehicle.



IoT Gateways & frameworks is a gateway to the internet for all the devices we want to interact with. Gateways act as a carrier between the internal network of sensor nodes with the Internet. They do this by collecting the data from sensor nodes and transmitting them to the internet.

In the **Cloud server,** the data transmitted through the gateway is stored & processed securely within the cloud server, i.e., in data centers. This processed data is used to perform intelligent actions that make all our devices. In the cloud, all analytics and decision making happen considering user comfort.

In **Mobile applications,** mobile apps will help end-users control and monitor their devices from remote locations. These apps push important information from the cloud on your smartphones, tablets. We can send a command to sensors to change the mobile application's values, like changing the air conditioner's default temperature.

B. IoT Pattern Classification

We identified three dimensions to classify IoT patterns: abstraction level, domain specificity, and quality characteristic.

Based on the Level of Abstraction, patterns are classified into the following three types.

- **High Level or Architecture Patterns** are often used at early phases such as analysis and architecture design.
- **Mid-Level or Design Patterns** are often documented as architecture patterns that encapsulate contexts, recurrent problems, and corresponding solutions.
- **Low-Level or Software-dependent Patterns** target specific modules or limited parts of the entire system and software. They regarded the abstraction level of the design patterns as low and are often used in detailed design and construction phases.

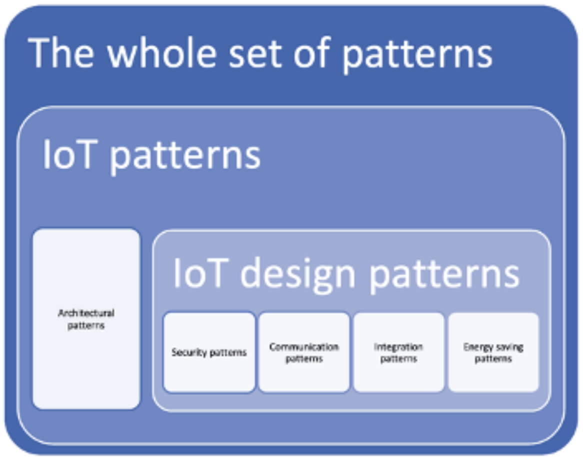
Based on Domain specificity, patterns are divided into three types.

- **Any Patterns** are general systems and software architecture patterns that can be adopted to design IoT systems and software if their contexts and problems match the patterns' contexts and problems.
- **General IoT design** patterns apply to any IoT systems and software.
- **Specific IoT Patterns** design patterns that address specific domain problems such as healthcare and brain-computer interaction.

All systems and software design patterns address some **Quality Characteristics**. IoT design patterns address interoperability, which is defined as a sub characteristic of compatibility. Additionally, there are other emerging characteristics like scalability and privacy.

C. IoT Design Pattern Classification

Design patterns are divided into the following groups: Integration patterns, Communication patterns, Security patterns, and Energy Saving patterns. By adding a group of Architectural patterns to the design patterns, a classification is obtained to detect the patterns when needed easily.



D. IoT Design Patterns

Architectural patterns determine how the system elements are organized, interacted, and suggest to-build a single system in a heterogeneous system. They introduce the essential characteristics and behavior of the system. Components are introduced into the system to create different types of integration and communication of its components. These include Device-to-Device, Networking-to- Networking, Middleware-to-Middleware, Application & Services-to-Application & Services, Data & Semantics-to-Data & Semantics (DS2DS). IoT Architecture Pattern provides a three-tier architecture with a User Interfacing Layer is an application layer that allows users to monitor and control devices remotely. Interconnection and Logic Layer is a networking and data communication layer that ensures the coordination and collaboration of devices across the network. Computation and Data Storage Layer manages intelligent analysis and data storage.

Integration design patterns ensure the way to combine devices located at different end nodes with different protocols in an IoT platform. These are low and general specificity patterns. To solve the physical infrastructure of IoT in different networks and devices, we use the following patterns.

- The **Edge Provisioning Pattern** provides control over a large number of hard-to-reach scattered end devices, enabling them to be easily reconfigured, as well as to incorporate new ones easily.
- The **Edge Code Deployment Pattern** provides the ability to easily maintain device software through decentralized Git version control located on the backend server. They deploy their code to many IoT devices automatically, quickly, and safely, and configure them without being concerned about the long process of build, deployment, test, and release. Example: Git
- The **Edge Orchestration Pattern** distributes functionality in such a way that it allows end nodes to control, configure and manage their own end devices, monitor their status, discover the services they need.
- The **Edge Diameter of Things (DOT) Pattern** requires the creation of a Metering Server that unifies the way how services are measured across endpoints, as different providers may use different usage patterns, such as event-based or time-based.

Communication design patterns are architectural concepts that describe how messages are transported in the network to accomplish certain devices' tasks. Knowledge of these patterns is important to correctly design and implement applications and connect in a network accordingly to meet functional and performance requirements.

Devices in the IoT can be categorized into groups according to their computational and communication capabilities.

- **Unconstrained Devices** have no significant constraints regarding their computational and communication capabilities.
- **Constrained Devices** are those constrained in their computation, storage, and communication.
- **Always-On** devices are those that stay connected and operational all the time.

Devices can operate in different **modes** depending on their communication frequency and their need to save energy.

- **Low-Power** devices usually communicate frequently. They will sleep for short periods between communicating but will generally stay connected to the network.
- **Normally Off** devices will be asleep most of the time and reconnect to the network at specific intervals to communicate

Some of the communication patterns are Device Wake-up Trigger Pattern, Device Gateway Pattern, Device Shadow Pattern, Rule Engine Pattern, Delta Update Pattern, etc.

The **Device Wakeup Trigger pattern** involves sending a message over a communication channel to a device control unit that is not permanently connected to the network. It is used to contact sleeping devices immediately.

Problem:

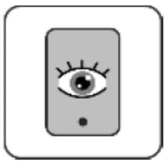
Some devices might go into a sleep mode to conserve energy and only wake up from time to time to reconnect to the network. During sleep, they are not reachable on their regular communication channels.

Solution:

Implement a mechanism that allows the server to send a trigger message to the device via a low energy communication channel. Have the device receive those triggering messages and immediately establish communication with the server when it gets a notification.

Energy-saving design patterns are gaining popularity nowadays. These patterns aren't related to software, and they only focus on communicating with devices to save energy.

The **Always-On Device pattern** refers to cases where energy saving is inefficient.



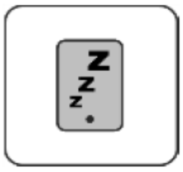
Problem:

You have a device with an unlimited energy supply and need to have it available and responsive at all times.

Solution:

Leave the device turned on and connected to power at all times.

The **Normally Sleeping Device pattern** is used when the device does not need to operate continuously and is implemented by deactivating all its energy elements.



Problem:

You have a device with a limited energy supply. You want to minimize the power used by the device.

Solution:

Program the device to disable its main components when they are not needed. Leave a small circuit powered, which reactivates the components after a predefined amount of time.

Security design patterns are designed to prevent accidental insertion of code vulnerabilities and mitigate these vulnerabilities' effects. They deal with authentication and authorization, as well as secure communication. Security patterns range from architectural patterns involving system-wide design to execution-level patterns relating to the system's implementation of individual functions or methods.

Some of the security patterns are Trusted Communication Partner, Permission Control, Outbound-only Connection, Personal Zone Hub, Whitelist and Blacklist, etc.,

Trusted Communication Partner

Context:

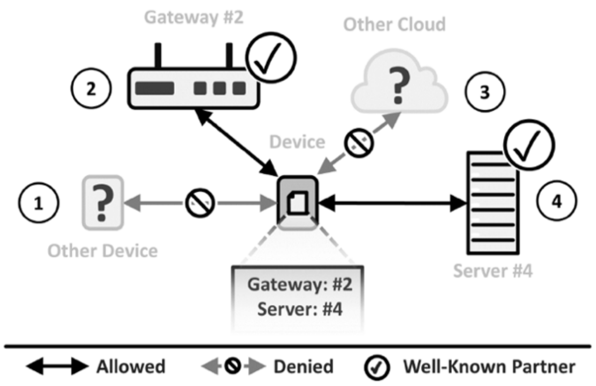
In the IoT, devices may communicate with many other communication partners, such as backend servers, applications, or other devices. Some of these connections may be used regularly, while others are used infrequently.

Problem:

In a dynamic environment, there may be multiple potential communication partners available for a device. These may not be known or trusted and may pose a security risk as attackers may use them to access devices and their networks.

Solution:

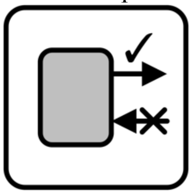
Each device is configured to communicate with a limited selection of Trusted Communication Partners placed on the device during bootstrapping. Trusted Communication Partners may be implemented in the form of single entries in a configuration file. On the device, a component checks each incoming and outgoing connection. If the connection source or target connections match the entry in the configuration file, it can pass. Otherwise, the connection is blocked and logged. For some use cases, it makes sense to only use Trusted Communication Partners for certain functionality.



Outbound-Only Connection

Context:

Devices send data to other devices for monitoring, analysis, or storage purposes. They also receive commands which control, or trigger functionality built into them. This communication poses a security risk as attackers might try to gain access to devices to manipulate them.

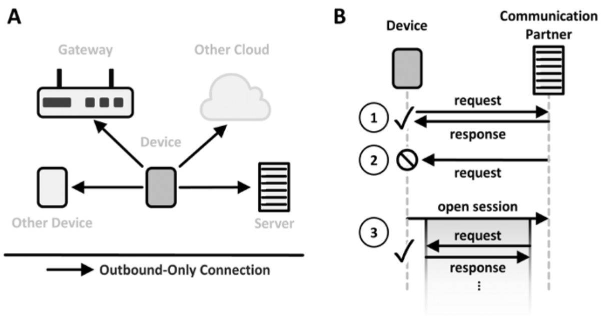


Problem:

Devices are a target for attackers who try to gain access to their network. They may send unsolicited communication requests to the devices to get them to connect to an infected communication partner or to misuse them.

Solution:

All communication partners should be stored in a configuration file or database stored on the bootstrapping device. The device may use certain events, such as sensor reading above a threshold or other parameters, to work out when communicating with others. It initiates the connection and sends & receives the messages from communication partners, as shown in 1 in fig B. Some of the incoming requests will not be answered, then those requests are denied as number 2 in fig 3B. Number 3 in fig 3B is a long-lasting session, which allows its communication partner to send messages as long as the connection remains active. Here, it creates a long-lasting session that helps its communication partner send messages as long as the connection remains active. Outbound-Only Connection does not restrict bidirectional communication once a connection is established, but it does limit the initial creation of a connection from the outside.



Permission Control

Context:

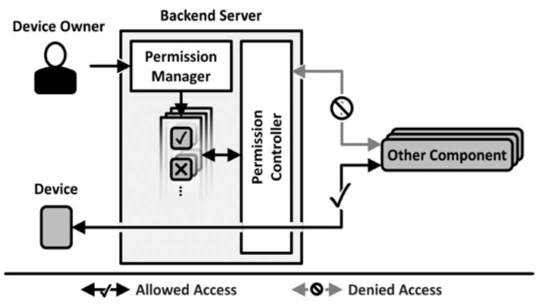
In the IoT, there are often multiple stakeholders involved, such as owners and users. Building and using IoT solutions usually requires communication between other stakeholders' components, like communication between devices and a backend server, as data and functionality are shared.

Problem:

Device owners are afraid to completely hand over access to their devices and data to third parties without any control. It is often unclear what data a device shares with communication partners or what others can access and control.

Solution:

Implementing Permission Control provides device owners with the means to control what external communication partners can do with a device. Before a device is added to the **Backend Server** or before another communication partner is allowed to access the device, the device owner is presented with a **Permission Manager** interface to explicitly allow or deny the communication partners some rights. The job of the **Permission Controller** is to grant authorization. Permission Control implements the existing Authorization pattern. Authorizations have to be enforced each time another component wants to access a particular device's functionality. Devices or components can change; every time any change occurs, you should require the user to confirm or alter his previously granted permissions.



Personal Zone Hub

Context:

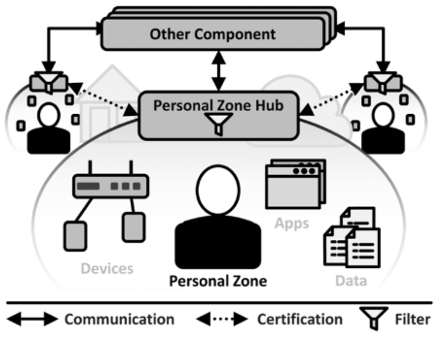
Devices usually have an owner, such as a person or an organization. Together, they create a personal zone for their owner. This personal zone may contain all kinds of devices, applications, and data that may allow others to identify, track, or gain insights about the owner. Thus, the owner is usually interested in controlling and restricting access to his zone.

Problem:

Users have an increasing number of IoT devices. Managing the permissions, data sharing, and control of these devices across multiple gateways and cloud systems is complex.

Solution:

A Personal Zone Hub acts as the entry point in coordinating a personal zone's agents. It encompasses the pool of digital resources belonging to one person. Conceptually it creates a logical boundary around a person and their devices, apps, services, and data. Technically, it combines multiple types of devices and networks, connected or disconnected, and cloud providers through a virtual network. The Personal Zone Hub controls access to the personal zone through a logical Firewall, where the owner may allow different parties to access all components or data. To address each hub uniquely, we use a routable URL by placing the Personal Zone Hub into a cloud, where it is accessible for other communication partners.



IV. Conclusion

All the design patterns which we mentioned here help ML developers and IoT architects to understand and build ML models and IoT products. These design patterns summarize specialists' experience and offer the ability to solve some of the system development problems. As ML and IoT are dynamically evolving, new design patterns are developed continuously to solve the problems; because of plenty of patterns, there is small confusion to choose the right pattern. Based on architecture layers and categories, we have proposed a group of patterns to use in a particular situation in development systems, focusing on method, architecture, and qualitative results.

REFERENCES

[1] Valliappa Lakshmanan, Sara Robinson, Michael Munn, "Machine Learning Design Patterns" (textbook resource).

[2] Ruchi Sharma, Kiran Davuluri, "Design Patterns for Machine Learning Applications", Computing Methodologies and Communication (ICCMC) 2019 International Conference.

[3] Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. 2016,2017 "Internet of Things Patterns and Internet of Things Security Patterns," HILLSIDE Proc. of Conference on Pattern Languages of Programs.

[4] Hironori Washizaki, Nobukazu Yoshioka, Atsuo Hazeyama, Takehisa Kato, Haruhiko Kaiya, Shinpei Ogata, Takao Okubo, Eduardo B. Fernandez, "Landscape of IoT Patterns," Software Engineering Research and Practices for the Internet of Things (SERP4IoT) 2019 International Workshop.

[5] Eduardo B. Fernandez, "Security Patterns in Practice: Designing Secure Architectures Using Software Patterns" (textbook resource).