CHAPTER 2

IoT Frameworks and Complexity

The complexity of things – the things within things – just seems to be endless. I mean nothing is easy, nothing is simple.

—Alice Munro[1]

Introduction

In Chapter 1 we explored device cost dynamics when security is built-in from the beginning. Either the cost of the device increases or the ratio of device resources attributed to non-security-related functionality decreases. However, ignoring security results in the IoT device becoming the "weak link." This chapter surveys IoT frameworks. We categorized them according to a consumer, industrial, or manageability focus though many seek broader relevance. IoT frameworks hide a lot of underlying complexity as the industry wrestles with embracing newer Internet protocols while maintaining backward compatibility. A plethora of standards setting groups have come to the rescue offering

[1] www.brainyquote.com/quotes/alice_munro_176434

S. Cheruvu et al., *Demystifying Internet of Things Security*,
https://doi.org/10.1007/978-1-4842-2896-8_2

insightful perspectives on framework design to accommodate broader interoperability goals. But this may be too much of a good thing as framework interoperability has become yet another interoperability challenge. Framework designs often emphasize differing objectives, interoperability, adaptability, performance, and manageability. We offer an idealized framework that focuses on security to add contrast to what the industry already has considered. This chapter is lengthy relative to the other chapters in part because there are many IoT framework standards available and each takes a different perspective. Each has merit but ultimately the IoT ecosystem is likely to reduce the number of viable frameworks. We nevertheless encourage continued IoT framework evolution that removes unnecessary complexity and places security by design at the center.

Historical Background to IoT

Before the "Internet of Things" became a commonly used term, embedded control networks used for real-time distributed control were known as process automation protocols, also referred to as fieldbuses. Fieldbuses are commonly used to implement SCADA (Supervisory Control and Data Acquisition) networks, building automation, industrial process control, and manufacturing control networks. These systems tend to be extremely complex and difficult to manage, especially over time as the number of system endpoints grows and the usages demanded of these systems increase. SCADA systems often involve connecting programmable logic controllers (PLCs), proportional-integral-derivative (PID) controllers, sensors, actuators, and supervisory management consoles, all connected through fieldbus protocols. But fieldbus technology isn't limited to a single protocol or even a small number of protocols. There have been more than a hundred fieldbus protocols entering industrial automation markets

in the last 20 years. The IEC-61158-1[2] and related standards describing fieldbus technologies contain over 18 families of fieldbus protocols. Some of these include CAN bus, BACnet, EtherCAT, Modbus, MTConnect, LonTalk, and ProfiNet. Wikipedia also has a fairly complete listing.[3] The Complexity can skyrocket when multiple fieldbus protocols are used to create an interconnected system. Then, with the birth of IoT, these fieldbus protocols are required to interconnect with Internet protocols, in some cases by replacing a fieldbus layer with an IP layer, which adds further complexity. When IoT systems are built to integrate with existing systems, based on fieldbus protocols, IoT systems are sometimes referred to as *brownfield* IoT because they represent use cases, ecosystems, and solutions that existed before the introduction of Internet technologies. Looking forward, industrial process automation and control, building automation, electrical grid automation, and automobile automation might continue using brownfield IoT nomenclature even though Internet technology integration is taking place.

Nevertheless, existing brownfield systems are highly proprietary and vertically integrated solutions, while Internet protocols historically have been more open and layered and support a richer ecosystem of vendors and value-added suppliers. Reducing fragmentation of brownfield networks through IT/OT convergence is a key motivation for IoT. Possibly it is this openness and richness of the Internet that drives the OT industry toward an "Internet of Things." Additionally, with respect to security, IT priorities have focused on CIA (confidentiality, integrity, and availability), in that order, while OT has prioritized availability and integrity above confidentiality. The tension between CIA trade-offs is an important consideration as the IT and OT come closer together.

---

[2]IEC 61158-1:2019 "Industrial communication networks - Fieldbus specifications - Part 1: Overview and guidance for the IEC 61158 and IEC 61784 series", International Standard, Ed. 2.0, 2019-04-10. Available at: https://webstore.iec.ch/publication/59890

[3]https://en.wikipedia.org/wiki/List_of_automation_protocols

Instead of using existing system as the starting point, the Internet of Things can bring a fresh perspective. Extending Internet connectivity beyond desktops, laptops, smartphones, data centers, cloud computing, and enterprise computing to agricultural, industrial, energy, health, transportation, public sector, and critical infrastructure seems a reasonable context for understanding the momentum behind the Internet of Things (IoT) evolution. The use of IoT technology to implement a completely new IoT system spawns unique applications for operational automation; building such a system with wholly new technology and protocols is sometimes referred to as *greenfield* IoT technology. Some examples may include drone control, self-driving cars, smart cities, supply chain automation, and machine learning. Greenfield IoT is riding the Internet wave of less-proprietary, lower-cost, and increasingly ubiquitous network technology that revolutionized PC, data center, and mobile device networks in the 1990s and 2000s. IoT may also benefit from the wave of microprocessor, memory, power, and storage innovations in mobile computing that results in lower-cost but highly capable computing platforms.

Whether the system is a brownfield system tying existing industrial or manufacturing automation control system with Internet technology or a greenfield system using completely new protocols and devices, both instances of IoT systems bring a level of intricacy that necessitates some abstractions to improve application development efficiency and to make management of these systems feasible.

But it isn't just the protocols that generate complexity in IoT systems. Industrial IoT systems may have multiple layers of networks connected through gateways. IoT systems may best be categorized as a *system of systems*. As security practitioners contemplating the prospect of securing a complex system of systems, we must take every opportunity to ask whether

the complexity is justified because we, like other security practitioners, believe complexity is the enemy of security.[4]

IoT Ecosystem

The IoT ecosystem is extremely complicated, fragmented, and evolving. It evolves at different rates depending on many factors, one of which is the replacement cycle for a given solution or industry. The replacement cycle for business PCs is 3–5 years, smartphone replacement is 1–3 years. Contrast this with building automation where an HVAC system replacement cycle is 15–20 years or nuclear power generation facilities that must replace failing parts with identical replacement parts – leaving no room for the introduction of innovative or more secure technologies. These refresh rates either speed adoption of new technologies or restrict, even inhibit, the adoption of technologies that might improve operations, reduce costs, or even protect lives.

Due to the many differences in various sectors of the IoT ecosystem (e.g., health, public, transportation, industrial, energy), the sectors appear to embrace Internet technology differently – in silos (refer to Figure 2-1). However, the market forces keeping the silos defined are due in part to the technical requirements unique to the usages and applications that drive internal market cohesion. Brownfield solutions may have benefitted from proprietary or vertically integrated solutions, aided by these cohesive market forces, long replacement cycles, and costly specialized hardware components. But that is unlikely to persist as IoT innovations continue to find technology adjacencies that spill over silo barriers causing technological disruptive innovation. Generally, this is a good thing. However, these disruptive forces breaking down the proprietary silos also brings new challenges that impacts security in the form of increased complexity, new business models and unanticipated interactions.

[4]Tom Gillis, Contributor, Network World, "Complexity is the enemy of security," Aug 8, 2016. www.networkworld.com/article/3103474/security/complexity-is-the-enemy-of-security.html

Just as the changes in Internet protocols brought more complexity to PC networks in the 1990's, Internet of Things technologies promise more complexity (at least initially) for industrial, control and automation systems.

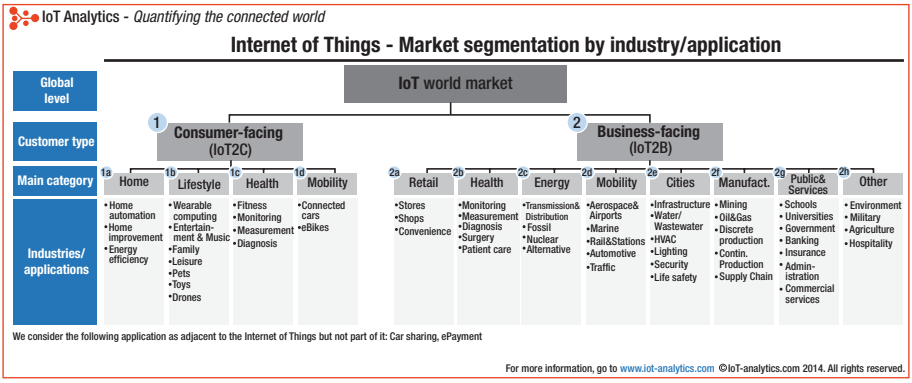

Figure 2-1. *IoT market segmentation by industry/application*[5]

The IoT ecosystem (referring to Figure 2-2) can be understood in terms of concentric rings of technology used to connect distributed physical and logical components. The technology within a particular ecosystem is specialized for that ecosystem, its business models, as well as the producers and consumers in that market. Ecosystem-specific components are specialized for different aspects of an ecosystem's distributed applications, resulting in unique devices that coordinate sensing, actuation, control, data collection, data aggregation, data analysis, risk management, and operations. IoT system components may be distributed because of physicality of sensing and actuating, or due to efficiency requirements that result in specialized computation. A potential unifying factor in all this is an interoperable, low-cost networking capability that makes distributed IoT possible. But satisfying the myriad needs canvasing multiple IoT segments using a single IoT technology seems improbable if not impossible.

[5]IoT Analytics, Knud Lasse Lueth, "IoT market segments – Biggest opportunities in industrial manufacturing," Oct 31, 2014. https://iot-analytics.com/iot-market-segments-analysis/

Connectivity Technology

Network and connectivity are nevertheless of paramount importance. IoT systems must enable connections over short-, medium-, and long-range distances. IoT solutions often must satisfy a wide range of transmission quality requirements that may also need optimizations for low latency, isochronous, asynchronous, store-and-forward, mobility, or streaming. IoT systems must consider environmental disturbances such as radio interference or emissions from other electronic equipment, low-power conditions, congestion, and resource starvation scenarios. Guaranteed service levels also add to the mix of requirements.

Additionally, trade-off decisions impact safety, reliability, resiliency, security, and availability. A variety of network technologies have emerged to address the multifaceted needs of IoT such as Zigbee, Industrial Ethernet, LoRa, LPWAN, Modbus, and TSN – to name a few. Some are highly specialized to a specific application context such as the Control Area Networks (CAN), which uniquely addresses the safety critical automated braking systems found in many automobiles. Fieldbus protocols, such as Modbus, use a synchronized communications bus to ensure each PLC (programmable logic controller) receives the messages directed at it.

While others are more general purpose such as Wi-Fi, Bluetooth, 5G, and Ethernet that accommodates information networks, streaming media, as well as control network applications. Industrial Ethernet operating at very high data rates can accommodate industrial real-time control requirements by ensuring network utilization remains below about 10%. Chapter 5 will dive deeper into details of different connectivity interfaces and considerations facing consumer and industrial IoT.

Messaging Technology

IoT frameworks are exposed to IoT applications using a data model abstraction. The framework data model describes a view of the network where nodes appear as flat or nested data structures, and updates to

data values may result in actuation of various controllable elements. The data model abstraction allows the applications to focus on capturing semantic richness and less on moving data from node to node. Data are represented as structured markup that easily maps to messaging transport technologies.

Messaging technology determines how messages flow between network nodes. It also facilitates the building of IoT systems that collect data from various nodes using disparate protocols at the expense of creating additional complexity in the messaging layer. Simple messaging is request-response based such as REST (Representational State Transfer). HTTP (Hypertext Transfer Protocol) and CoAP (Constrained Application Protocol) follow the REST methodology. Publish-subscribe messaging allows multiple nodes to register for notifications when a change is detected in a variable on a peer node. MQTT (Message Queuing Telemetry Transport) is a popular example of a publish-subscribe messaging system. Broadcast and multicast can make publish-subscribe more efficient, which may be used in some IP-based networks. Different protocols are useful in different environments, and the whole communication stack even down to the availability of broadcast at the network physical layer must be considered when developing services in an IoT system. This complexity is difficult for the system designer but becomes overwhelming to the IoT developer. This complexity becomes most evident when designing an IoT platform, especially when designing an IoT platform intended to service multiple ecosystems. Platforms manage this complexity through the use of IoT frameworks.

Platform Technology

IoT platforms host applications, resources, and data useful to an IoT distributed application. Platforms are specialized to the type of work each performs. Constrained IoT platforms may optimize for connectivity, latency, and small footprint, while less constrained platforms at the OT

network edge may optimize for device offload and bridging across control domains. Cloud platforms optimize for compute, scalability, capacity, and analytics. IoT frameworks are used in platforms because they facilitate interoperability and connectivity by combining appropriate networking, protocol, and platform ingredients in ways that allow application portability regardless of the node's native specialization characteristics.

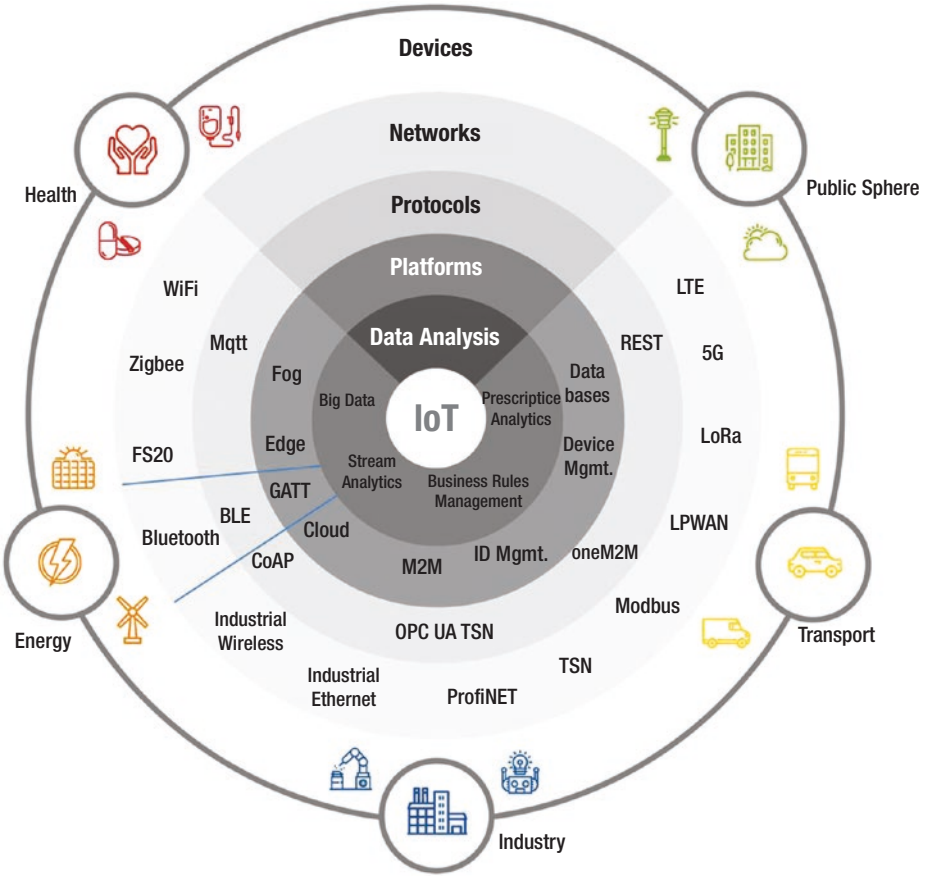

Figure 2-2. *IoT ecosystem*

Elements of an IoT System

This section describes the elements of an IoT system focusing on device architecture, network architecture (an interconnected collection of devices), system management architecture, and lastly framework architecture.

IoT Device

The term "device" can be confusing because it means different things in different contexts. When viewed from a manufacturing perspective, the device is a physical component consisting of hardware, firmware, and system software. It may also be preloaded with application software compiled into a single image that is embedded into persistent memory.

When viewed from a network management perspective, a device is a node that has a network address and could be part of a collection of interconnected devices. There could be multiple network endpoint addresses per physical device. Furthermore, given multiple network interfaces, the same physical device could appear as multiple nodes to other devices.

When viewed from an IoT framework perspective, a device is a logical context that exposes message passing interfaces. Interfaces are used to exchange data that is structured according to an interface definition. The actual data structure as viewed from within the framework may differ depending on the network protocols, message passing technology, or system usage. A logical device may have multiple interfaces to the network giving the impression to peer nodes there are multiple physical devices. This can be confusing if network address is the only way to disambiguate IoT devices. IoT frameworks expose a logical IoT device whose identity is independent of the underlying connectivity layer. However, security challenges can arise when a single networking interface exposes shared data or control surfaces with multiple logical

devices. This creates an opportunity for an attacker to exfiltrate data, perform side-channel analysis, or maliciously control logical devices. Consequently, the security design should incorporate endpoint protection technology deeper into the system – at the logical device level.

When viewed from an application perspective, the IoT framework data abstractions can make it difficult for application code to tell when a physical device boundary is crossed. A single application may interact with multiple IoT framework "devices" not knowing if they are geographically local or remote. This is relevant to security practitioners because device physicality is often what defines a security boundary. Obscured security boundaries make it more difficult for applications to effectively apply security protections.

To avoid confusion, the authors try to provide clarifying context whenever "device" terminology is used.

IoT Device Architectural Goals

Unlike smartphones, PCs, laptops, and servers, the device bill of materials (BOM) for constrained IoT devices is often under significant cost pressure. In addition to the expected processing requirements, IoT devices often must accommodate hostile operating conditions that include extreme temperatures, vibration, humidity, and ultraviolet radiation. Meeting BOM constraints implies every ingredient is scrutinized to identify the minimum viable hardware, firmware, and software configuration while still satisfying product requirements. Part substitutions may be made over the course of a product's lifetime to lower production costs.[6] The IoT supply chain competes to be the low-cost supplier, and device vendors want to foster this competition to drive component costs even lower. Common interfaces facilitate interoperability and the integration of specialized hardware with

[6]Vendors often qualify multiple suppliers for hardware components that perform essentially the same function but allow production lines to keep producing if one supplier's supply chain happen to be disrupted.

general purpose hardware, sensor, accelerator, and Field Programmable Gate Array (FPGA) processor integration traditionally is done by a device manufacturer, but increasingly, specialized functionality is exposed to the network as a service. Software layers create logical devices that may be dynamically defined. Software defined devices offers greater flexibility for tailoring IoT solutions that meet customer need. Securing software defined devices requires a trusted execution environment that creates trustworthy hardware isolation and exposes security roots of trust to the soft device.

Interoperability

Architecting a device to be interoperable with other devices or infrastructure already, or soon to be, on the market is of paramount importance for IoT, especially given the enormity of different devices in large IoT systems. Web-based validation suites allow device vendors to verify their products will interoperate with a wide variety of other vendors' products, which would be too numerous to exhaustively validate using direct interactions from device to device. Testing for interoperability with an actual device that has not completed development or is not yet released to market is simply not possible. However, web validation suites allow testing for interoperability with standard protocols and frameworks, ensuring compatibility with peer IoT devices that have not yet completed development.

Nevertheless, interoperability gaps are likely to exist. For example, data models developed by competing standards may have syntactic differences even though semantics are similar. Standard protocols may not fully interoperate if certification testing is missing or is not comprehensive. Simulation tools that virtually deploy customer-specific configurations can be helpful. Simulations help expose interoperability gaps in specifications and validation suites relating to software behavior and data definitions. Trial deployments and test beds are another technique for finding gaps. This helps find hardware-dependent incompatibilities. Trial deployments go live once the gaps can be corrected. Test beds can be used for

longer-term evolution of products with sequenced rollout of increasing capabilities and features while ensuring that interoperability or backward compatibility problems do not creep in.

It is prudent for IoT system designs to anticipate having to work around incompatibilities and building specific features into their design to compensate for such issues. Postdeployment reconfigurable layers between applications and embedded components give systems architects the ability to make corrections during simulation and trial deployment. Less constrained devices such as hub controllers, bridges, and gateways more easily accommodate reconfigurable layers as they often support a wider variety of network interfaces and have more computing resources and storage to draw upon. Nevertheless, reconfigurability comes with a security cost. Malware might more easily exploit reconfigurability features that compromise embedded system components.

Security

Security consists of both functionality and assurance disciplines. Security functionality typically deals with secure boot, secure key storage, and cryptographic algorithm acceleration, while security assurance typically deals with ensuring security functions work the way they are intended. Trusted computing technology combines security functionality with security assurance mechanisms so that security compromise isn't catastrophic. Trusted computing components are called upon to perform recovery steps. All devices contain some set of trusted functionalities, upon which all other parts of the system assume is trustworthy and has not been compromised; this is called the *root of trust* for the device. The root of trust is normally involved in the secure booting of the device, holding the device's identity credentials, and presenting cryptographic evidence of device claims, called attestation. Depending on the device, the quality of the root-of-trust may vary.

In less constrained environments, a root-of-trust could be a security subsystem such as a Trusted Computing Group (TCG) Trusted Platform

Module (TPM) or a secure storage module such as Replay Protected Memory Block (RPMB). It could be a secure coprocessor such as ARM TrustZone or a security mode of a CPU such as Intel Software Guard Extensions (SGX). All other software and hardware components depend on the root-of-trust components in some way for their security.

Typically, less constrained systems make use of multiple roots-of-trust and multiple trusted execution environments. For example, trusted boot may rely on a root-of-trust for measurement in the form of a boot ROM that computes an integrity value for software images loaded during boot-up. These integrity values are stored in another root-of-trust for storage that protects them until they're queried by a remote device that verifies boot integrity. The remote device expects to receive an attestation report that is signed by a trustworthy signing key protected by a root-of-trust for reporting. The TPM is an example of a discrete processor that combines roots-of-trust for storage and reporting.

Roots-of-trust can protect application code while it executes using Trusted Execution Environment (TEE) technology such as Intel SGX. Application developers partition application functionality according to the functions that are security relevant and those that aren't. Less constrained environments allow multiple TEE instances. Managing and deploying multiple trusted environments and roots of trust adds cost and complexity.

In more constrained devices, these costs may be too high. Instead, devices must be designed with layered trusted computing. The Trusted Computing Group (TCG) proposed an approach for secure constrained device boot, secure device identity creation, and device attestation (Figure 2-3) that doesn't depend on a security coprocessor called Device Identity Composition Engine (DICE).

Using a DICE strategy, the root-of-trust elements are those that operate first when the device is reset or when it resumes from a nonoperational state. The DICE architecture defines a Unique Device Secret (UDS) that is a circuit that produces a unique number when the platform undergoes power reset. The UDS circuit reads low-level device firmware

that is used to boot and possibly operate the device once booted. Firmware is cryptographically hashed with the UDS that is then fed into a cryptographic key generation circuit to produce a device identifier. Cryptographic hash is a one-way function that ensures input data can't be discovered by analyzing the output value. If a different firmware image is hashed, it will produce a different hash output value. This will cause the key generation circuit to produce a different device identifier than what results from the first firmware image. If the device identity changes from what the IoT network expects, the changed device identity is no longer trusted and must be revetted and onboarded into the network.

The device identifier is unique to the UDS secret and the firmware installed. The secret is immutable because it is hardware. If the firmware is updated, a different device identity key is generated. A controller, bridge, gateway, or other IoT nodes can determine if firmware changes because it will no longer recognize the device identifier or be able to verify its digital signature. If malware corrupts device firmware then resetting the device sill return it to a secure operational state. The UDS and DeviceID derivation functionality form a root of trust that is simpler than a traditional Trusted Platform Module (TPM), secure co-processor or TEE. This is better suited for cost constrained IoT devices, but also benefits TCB design by tailoring TCB functionality that is most appropriate for special purpose IoT devices.

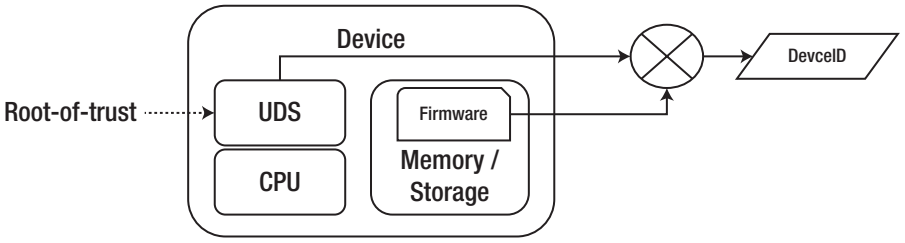

Figure 2-3. *Device Identity Composition Engine*

IoT Network

When multiple IoT devices are connected together, they form an IoT network. However, connectivity alone isn't very interesting. IoT devices should interoperate as a distributed application. One expects IoT nodes will cooperate to achieve a common objective. To do this, devices need a few basic behaviors: (a) the ability to discover peer nodes, something about their function or role and interfaces they support; (b) the ability to connect, which may involve authenticating and constructing a secure channel or cryptographic association; and (c) the ability to send and receive formatted data, parse it, and process it according to application-specific semantics.

Core to IoT design is the idea of an hourglass network layering model (Figure 2-4) that seeks to simplify the possible choices of network layer protocols to Internet Protocols (IPv4 and IPv6) while permitting legacy SCADA, fieldbus, and embedded control physical and data link layer technologies to remain available either through gateways or through encapsulation, such as 6LoWPAN[7] (IPv6 over Low-Power Wireless Personal Area Networks).

The top half of the hourglass hosts existing and evolving IP transport layer technologies, for example, the Constrained Application Protocol (CoAP)[8] supports an HTTP-like RESTful message exchange without the overhead required to support HTTP and TCP. The Datagram Transport Layer Security (DTLS)[9] applies TLS-like security to CoAP. An impressive array of emerging protocols designed for IoT are being developed by the IETF Constrained RESTful Environments (CORE)[10] working group. DTLS may be appropriate in cases where reliability and in-order guarantees are not needed.

---

[7] https://tools.ietf.org/html/rfc4944

[8] https://datatracker.ietf.org/doc/rfc7252/

[9] https://datatracker.ietf.org/doc/rfc6347/

[10] https://datatracker.ietf.org/wg/core/documents/

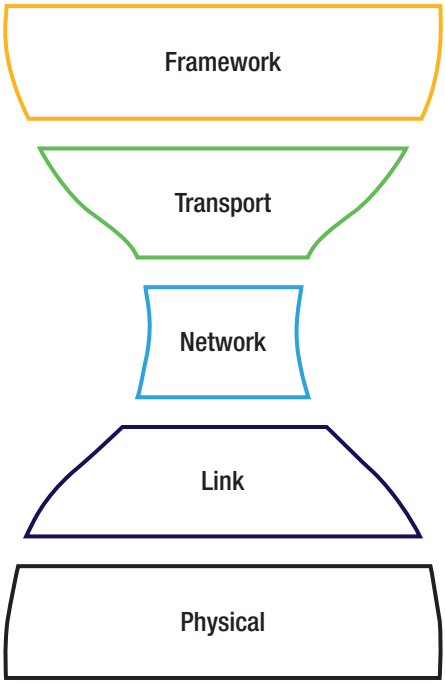

Figure 2-4. *IoT network layering*

The framework layer sits atop the hourglass consisting of a dizzying mix of technologies that predate IoT or have emerged as a result of it. Most interestingly a flurry of new standards organizations has emerged that seem to have insightful perspectives on how best to define IoT frameworks. The authors believe that much of the IoT ecosystem will coalesce around a common set of Internet-based technologies forming an hourglass shape.

IoT System Management

IoT system management comprehends manageability goals for both IT (Information Technology) and OT (Operational Technology). Device lifecycle management is common to both IT and OT disciplines covering the full spectrum beginning with manufacturing and supply chain through

all phases of operation, including decommissioning and retirement. Management services support device lifecycle management. These include security services for managing roles, access control policies, and cryptographic keys and certificates; software update services for distribution and installation of firmware, software, and security patches; orchestration services for coordinating distributed application behavior, simulation, and for handling graceful failover, resiliency, load balancing, and redundancy; and telemetry services report on a variety of operational, security, safety, and behavior components of an IoT system that may be used further by IoT analytics and business management.

A challenge facing IoT systems is finding a uniform and consistent approach to manageability given the deeply fragmented brownfield and greenfield IoT solutions. Proprietary and vertically integrated solutions often don't interoperate with horizontal IoT framework approaches, and framework manageability is quite often rudimentary lacking deep integration.

Lack of a uniform approach to security manageability has potentially significant IT and OT impact. For example, application of a security patch in an industrial IoT deployment may require multiple security consoles with labor-intensive checklists that verify all nodes are patched properly. Access control policies may not be consistently expressed across disparate IoT systems where role names and syntax may differ, access enforcement conventions may differ and be inconsistent, or key management capabilities may differ and may lack scalability or equivalent security strengths. Security gateways may be considered as a way to address some of these issues, but they may require deployment of new trusted nodes in situations where trust semantics don't normally expect or allow a universally trusted gateway system. For example, a security gateway node that links an industrial process automation network to a business analytics server might be located at a base station in a wireless edge environment that has limited physical security, but nevertheless must operate with full security privileges of both networks.

Device Lifecycle

Trust in logical IoT devices is (or should be) tied to trust in the physical layer that hosts it. In an enterprise deployment scenario, servers, PCs, and even smartphones can undergo a rigorous manual inspection and configuration step by trained security professionals. However, the scale at which IoT devices are deployed is seldom feasible to apply the same rigorous manual processes. Instead, onboarding techniques that require minimal or no touch are needed. IoT platforms and devices follow a lifecycle (Figure 2-5) that may begin during manufacturing and ends when the device is decommissioned or waterfalled to another owner for redeployment starting another lifecycle.

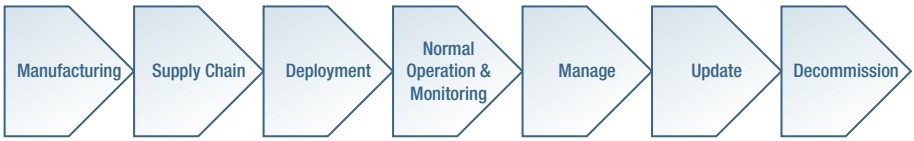

Figure 2-5. *IoT device/platform lifecycle model*

Attackers may target vulnerabilities earlier in the lifecycle in order to avoid detection and circumvent mitigation strategies that presume manufacturing, supply chain, and onboarding steps are free from compromise.

IoT frameworks make assumptions about where along the device lifecycle continuum the framework abstraction models begin to apply. Early in the lifecycle, only physical devices exist. Even if logical devices come into being early in the supply chain, it may still be possible for additional logical devices to appear subsequent to initial onboarding or may disappear prior to a final decommissioning step. Security of the IoT system may depend on how well the IoT framework layer integrates with the platform lifecycle.

Manufacturing

Manufacturing processes are critical toward the establishment of hardware-roots-of-trust which is a term used to describe security building blocks having to do with establishing platform/device identities, protecting cryptographic keys and algorithms, and creating hardened execution environments and system bootstrap procedures that resist attacks. Features may include hardware random number generation, cryptographic algorithms in ASICs (Application-Specific Integrated Circuit), FPGA (Field Programmable Gate Array) or instructions, hardware fuses that seed random number generation, boot ROM, replay protected memory, and others.

Supply Chain

Supply chain processes protect platforms and devices as they make their way from manufacturers to retailers to customer first deployment. Supply chain participants may have physical access to hardware components that if replaced by malicious components could result in undetectable attack scenarios. Tracking platform and devices through the supply chain may involve the use of RFID (Radio-Frequency Identification) tags, supply chain UUIDs (Universally Unique Identifiers), or cryptographic device identifiers. Privacy may become a challenge however as tracking capabilities could be misused in ways that violate privacy goals. Privacy requirements need to be anticipated as part of supply chain tracking mechanisms.

Deployment

Deployment is concerned with initial power up, customer-specific configuration, and establishment of the platform/device owner.
Then the entity responsible for adding IoT devices to their network is sometimes called the “owner” which implies a change of ownership and establishment of a “local” identity that differs from a manufacturer or

supply chain supplied identity. The owner operates onboarding services that facilitate ownership transfer, verification of supply chain provenance, attestation of security properties and roots of trust, issuance of credentials, security associations, roles, and access control policies. Taking ownership of many devices can be challenging given limited human resources and large numbers of devices. Zero-touch commissioning is immensely important and difficult to get right given the diversity in supply chain and given the spectrum of customer security and privacy expectations.

Normal Operation and Monitoring

Normal operation refers to operational states where IoT functions are fully enabled and ready for use. Security monitoring ensures devices and networks continue to function securely. IoT frameworks may choose to hide security monitoring operations from IoT application-level abstractions, but they should consider how to fail gracefully when security conditions require service disruption.

Manage

IoT devices require periodic management, tuning, and adjustment. Some management functions can occur while devices are operating normally. For example, addition of security credentials for dynamically added devices may not need to interrupt activity with existing devices. Other management tasks may require disruption of normal operations. For example, an uncalibrated actuator may result in device, process, or system failures if asked to operate outside its design constraints. Frameworks can facilitate communication of device status and availability to enable periodic maintenance without major disruption to peer nodes. This management implementation could be in-band (within the OS/FW control) and/or out-of-band (outside of OS/FW control).

Update

Software and firmware updates are arguably a subset of device management commonly known as Software Over-the-Air (SOTA) and Firmware Over-the-Air (FOTA) updates. Software update management must consider trade-offs of propagating large image files over networks optimized for small messages that may be latency sensitive. IoT networks may have "sleepy" nodes that are not available to receive an update in a timely manner.

Nevertheless, software and firmware updates are essential to secure operation. It is inevitable that security weaknesses will exist in most firmware and software images. Hence, when weaknesses are found, they should be fixed quickly to avoid possible exploit.

Decommissioning

Decommissioning is the process of undoing onboarding, commissioning, and provisioning that were applied previously. Although it is expected that devices and frameworks will anticipate scenarios involving devices that don't go through a decommissioning process to handle it gracefully, applying decommissioning steps helps ensure privacy objectives are met by removing trackable personally identifiable information (PII) or privacy-sensitive information before it falls into other hands. Decommissioning also ensures security-sensitive data, credentials, keys, and access tokens are removed so they aren't used to later attack other nodes. Frameworks can facilitate decommissioning by orchestrating the nodes removal in a coordinated way. Sometimes decommissioning could entail replacing the device under consideration with another device consisting of the same persona.

Automation of the IoT device lifecycle is an important security capability as it helps ensure the device never enters an insecure state and minimizes opportunities for attacker exploit by ensuring secure lifecycle practices are consistently applied.

IoT Framework

An IoT framework is a middleware layer beneath one or more IoT applications that presents a network-facing application interface through which peer framework nodes interact. Frameworks often support multiple communication technologies and message passing techniques. IoT frameworks also expose security capabilities including hardware-roots-of-trust to applications and peer framework nodes.

IoT Framework Design Goals

IoT frameworks have four primary design goals: (1) reduce development time and bring IoT solutions to market sooner; (2) reduce apparent complexity of deploying and operating an IoT network; (3) improve application portability and interoperability; and (4) improve serviceability, reliability, and maintainability. Given the vast range of existing and emerging communication technology choices, it is untenable for applications to manage the combinations of possible ways to connect. Frameworks hide connectivity complexity beneath a higher-level message passing abstraction like REST and publish-subscribe. Standards organizations help achieve these goals through standardization of the framework layer interconnect, message passing interface definition, and data definitions leveraged by applications. Standards groups also document IoT system design principles, architecture, and interconnect options. Standards organizations and industry consortia may assist developers by supplying and certifying reference implementations that include source code. Reference code helps streamline development by providing implementations that pass compliance tests and correctly interprets standards specifications. Reference codebases are easier to maintain benefiting from a large diverse community of open source developers who cooperate by actively developing code and improving the codebase.

Frameworks simplify IoT networks by creating an abstraction of the IoT device networks that hides much of the underlying complexity while exposing data, interfaces, and functions that facilitate interoperation. All it should take to develop an IoT application is to create an application in a high-level language such as Node.js that utilizes framework APIs. The framework provides a semantically rich description of IoT nodes, objects, and interactions that allow IoT network designers to focus only on node interaction semantics rather than on the details of connectivity.

Frameworks facilitate improved application portability. This can be achieved at different levels. The bottom layer of the framework is operating system specific. The top layer of the framework is IoT use case specific in that it exposes a data model abstraction that reinforces an IoT usage context. Some examples include lighting control, home automation, health monitoring, entertainment, process automation, industrial control, and autonomous control. IoT applications can be developed once given the framework abstraction and can execute on any OS the framework is ported to. The details of dissimilar OSs and platforms can be hidden where porting of framework code to another OS (source code–level compatibility) can happen independently of application development. Binary compatible platforms can migrate compiled framework code across platforms using the same binary. Platforms that are not binary compatible may rely on virtualization to host framework images or may rely on device management services that hide the complexity associated with paring and installing the right framework with the correct platform.

Frameworks enable interoperable devices in heterogeneous environments. Consider a hypothetical scenario where devices are running different OSs and HW platforms. These devices could be built by different platform vendors using silicon from multiple vendors running different OSs such as Windows IoT Embedded and VxWorks running different middleware stacks. This is a perfect storm scenario for an IoT network deployment where there are too many possible combinations of connectivity and message exchange options to expect

speedy deployments. IoT frameworks come to the rescue by building the connectivity intelligence into the framework – hidden from application view and simplified from the device and network management view.

Frameworks also facilitate seamless manageability and serviceability by leveraging the framework's infrastructure to expose platform status information through the framework layer in accordance with the framework's data model abstraction. For example, a firmware update availability notification may be easily propagated across an IoT network. If the framework supports applying a firmware update, either push or pull, the firmware update images may be distributed over the air using the connectivity solution worked out by the framework.

IoT Data Model and System Abstractions

IoT frameworks define an application layer abstraction so that applications interact directly with framework data. For example, a temperature sensor might show the current temperature (*currTemp)* and the average temperature over the course of 24 hours (*aveTemp)*. Temperature values might be shown in Fahrenheit and Centigrade. Consequently, a data model description might be as follows:

```
{
    "tempSensor" = "/myTempSensor",
    {
        "currTemp"="85",
        "aveTemp"="70",
        "degrees"="Centigrade"
    }
}
```

Data modeling languages are used to richly describe framework objects according to a schema definition. Examples of data modeling languages include XML (eXtensible Markup Language), JSON (JavaScript

Object Notation), CBOR (Concise Binary Object Representation), and YANG (Yet Another Next Generation language) – just to name a few.

Data structures are accessed through well-defined network interfaces. For example, CoAP is a REST model interface that uses four methods: GET, PUT, POST, and DELETE to interact with framework data. A couple RESTful interface definition languages include RAML (Restful API Modeling Language) and Swagger.[11]

A framework node may consist of several objects such as a temperature sensor, camera, and light bulb. A *deviceId* may disambiguate multiple instances of a framework node. For example:

```
{
    "nodeType"="myDeviceType",
    "deviceID"="<UUID>",
    {
        "tempSensor" = "/myTempSensor",
        "ptzCamera" = "/myPtzCamera",
        "lightBulb" = "/myLight"
    }
}
```

Using these simple but powerful data modeling tools, IoT frameworks can describe elaborate IoT systems while hiding much of the network complexity that underlies connection establishment, routing, packet transmission, network address translation, and so on.

To a certain extent, IoT frameworks can be compared with Information-Centric Networking (ICN).[12] ICN rethinks the network where named information is the centerpiece of network architecture. Rather

[11] https://swagger.io/

[12] https://irtf.org/icnrg

than focusing on nodes, network topology, and protocol layering, ICN focuses on end-to-end data interactions. Data doesn't necessarily *reside on endpoints* but may be cached and replicated anywhere in the network. Like ICNs, the upper layer of IoT frameworks presents a data-centric view of the network. However, unlike ICNs existing protocol layering is retained. Arguably, this adds additional complexity but offers greater interoperability. Indeed, an ICN connectivity plugin to an IoT framework is a reasonable approach to bridge ICN with legacy networks.

Securing IoT messages must take an end-to-end view so that authentication, confidentiality, privacy, and authorization goals may be realized. Otherwise, the benefits of hiding complexity beneath an IoT framework may instead be hiding security gaps. The IoT application using an IoT framework may not be aware when security is managed using system layer interfaces. Internet protocols often have a secure alternative such as *https* for *http* and *coaps* for *coap*, where the "s" means security. A REST GET message works the same over *coaps* as it does for *coap*. The main difference is the Transport Layer Security (TLS) binding to the REST messaging protocol negotiates a secure session using credentials (keys and certificates) that may have been provisioned directly into the TLS subsystem without coordination through the framework layer. The framework layer may not be aware of the impact to authorization which can result in the framework misrepresenting actual security posture to IoT applications. IoT frameworks can differ significantly in their design and implementation attention to end-to-end security. We hope to illustrate this point more profoundly as we walk through a variety of IoT frameworks later in this chapter.

IoT Node

IoT frameworks define a *device* abstraction that is a logical representation of a physical device. This chapter uses the term IoT *node* to refer to the logical abstraction to avoid confusion regarding the physical device. Frameworks can create some interesting properties regarding IoT nodes:

- They may expose multiple nodes per framework to give the appearance of many nodes having the same IP address.
- They may consolidate multiple network addresses terminating into a common framework node.
- They may host services and capabilities that are dynamic – being created and deleted according to RESTful messages.
- They may impose system partitioning semantics such as dividing nodes into domains, groups, rooms, or some other semantic overlay.

Nevertheless, security semantics must remain true despite the framework abstraction. For example, if the node describes the endpoint where access is controlled, data is encrypted and decrypted. Then protection of the physical endpoint resources should strongly correlate with protection of the framework node.

IoT Operations Abstraction

IoT operations consist of several node interactions facilitated by frameworks. These include discovery, message exchange, event registration, and asynchronous notification. IoT nodes typically are not preconfigured to recognize other nodes. They must instead be discovered.

Discovery allows other framework nodes to inquire regarding supported interfaces and data structures essential to interoperability. Discovery can take many forms. For example, multicast and broadcast networking supports unsolicited discoveries. Nodes monitoring the broadcast may be required to disposition discovery events even if there is no action needed. Devices with limited battery capacity may have shorter life expectancies if deployed in highly dynamic networks. Alternatively, discovery may be accomplished by sending discovery requests to discovery interfaces for specific nodes querying the relevant information. This approach minimizes unnecessary activity on nodes that wouldn't otherwise need to participate. However, this approach may require multiple "drill down" discovery requests before finding the data or interface needed. Passive discovery employs directories or less constrained nodes that respond in place of other nodes that may disregard all discovery requests while in a low-power mode. The directory nodes satisfy the discovery phase so that power-constrained nodes only process the functions that they uniquely provide.

Discovery conventions:

- Consulting a directory of framework devices to learn device identities and how to connect – conceptually similar to LDAP (Lightweight Directory Access Protocol) commonly used by PCs in IT networks to accomplish a similar objective
- Inspecting a schema describing interfaces to learn which REST, publish/subscribe, and asynchronous notification messages can be used
- Querying the device directly to introspect its current state and configuration

Note An anonymous entity may learn a tremendous amount about how an IoT network functions, the type of nodes involved, what work they're capable of performing, and typical interaction patterns simply by using available discovery mechanisms. Given a small amount of additional information that links actual devices or users to the observable network, it may be relatively easy for an attacker to obtain or infer knowledge that otherwise is expected to be privacy sensitive.

Message exchange conventions:

- Preparing a message body whose syntax satisfies a recognized (standardized) data model schema
- Protecting the message using the appropriate security credentials
- Sending the message following the interface definition schema for the target node
- Collecting and processing the response message that similarly follows these conventions

Event handling conventions:

- Identifying objects and attributes available for participating in asynchronous events and conditions to be met that result in notifications.
- Preparing and sending a registration/subscription message following messaging exchange conventions.
- Maintaining context for processing asynchronous notifications.

- Nodes managing registrations/subscriptions must maintain context for secure delivery of the notification message(s) potentially involving many subscribers. Asynchronous message delivery may involve different security associations and context from those used to process registrations/subscriptions.

Connectivity Elements

IoT frameworks facilitate connectivity, gatewaying, and bridging. The following briefly summarizes how each is facilitated:

- Connectivity: Framework endpoint abstractions are mapped to network layer addresses and protocols where framework message exchange abstractions map to protocol specifics such as MTU (Maximum Transmission Unit) framing, multicasting, broadcasting, and packet delivery mechanisms.
- Gatewaying: Framework domain abstractions impose operational context for domain-specific filtering (hiding) traffic and performance of administrative duties.
- Bridging: Due to the proliferation of framework solutions, it is often necessary to translate from one framework environment to another. Framework bridging may have side effects where objects, interfaces, or semantics in one environment don't exactly translate to a second.

Manageability Elements

IoT frameworks may expose manageability elements through the framework object abstraction layer as a way for other framework objects and resources to better manage and respond to change resulting from management activity. However, this is more the exception than the rule. Even among horizontal open standard frameworks, there are many examples of device vendors wishing to retain proprietary or exclusive control over firmware/software update, onboarding, and cloud access capabilities. Nevertheless, frameworks can facilitate updates occurring outside the IoT framework by informing other nodes regarding pending updates or notifying regarding changes to version information. Additionally, IoT frameworks may not allow the framework itself to be updated from within an IoT framework context.

Security Elements

IoT frameworks need to accommodate security by ensuring endpoint nodes and their physical equivalents (i.e., device, process, virtual machine, enclave) have a secured identity, protected cryptographic keys and appropriately provisioned roles, credentials, and access policies. Endpoint security capabilities should protect sensitive data that is stored, transmitted, or manipulated locally outside of the IoT framework. Software and firmware should be protected when transmitted, installed, stored, and loaded for execution. Framework processing of encrypted data, access control decisions, and identities should be protected within an appropriately hardened Trusted Execution Environment (TEE) or isolated from non-framework aware services and interfaces. IoT device roots of trust should be used to protect device identities and ensure the appropriate firmware and software is loaded and executed.

Inherent to distributed systems is added risk associated with a dependence on multiple peer nodes that contribute data, processing, and administration to an overarching distributed application. Nodes

largely trust peer nodes to be in a correct operational state. However, this assumption of trust may not be justifiable without taking additional precautions to prove and verify the hardware, firmware, software, and operational state to peer nodes. Attestation is a security concept that addresses this concern but only if it is correctly implemented and integrated.

Consider the Cost of Cryptography

IoT systems are inherently distributed. Cryptography is an essential security building block technology for distributed systems. Nevertheless, cryptography imposes additional overhead in terms of computation, memory, storage, network bandwidth, and hardening. Symmetric cryptography generally speaking is lighter weight than asymmetric cryptography, and asymmetric cryptography is lighter weight than certificate-based asymmetric cryptography. IoT devices typically are designed with cost targets that may impact device cryptographic capabilities. Since these choices also impact interoperability, IoT frameworks must anticipate common cryptographic algorithms, key sizes, and key management infrastructures. Asymmetric cryptography is dominated by at least two algorithms: elliptic curve cryptography[13] (ECC) and Rivest-Shamir-Adelman (RSA)[14] algorithms. ECC has smaller key sizes than the RSA. ECC can accomplish the same level of security as RSA with key sizes that are 10–15% smaller. Key size is an important factor for constrained platforms as such many IoT standards require ECC.

Table 2-1 details some of the trade-offs associated with cryptography.

[13] https://tools.ietf.org/html/rfc6090

[14] https://tools.ietf.org/html/rfc8017

Table 2-1. *Trade-Offs Associated with the Type of Cryptography Used*

Criteria	Symmetric (Preshared Secrets)	Asymmetric (Raw Public/ Private Keys)	Asymmetric (Certified Public/ Private Keys)
Hardware Acceleration	Not Required	Required	Required
Memory Size	Small	Medium	Large (certificates)
Code Size	Small	Medium	Large (certificate parsing)
Message Size	Small	Medium	Large (certificates)
Persistent Storage Size	Small–Medium (depends on network size)	Medium–Large (depends on network size)	Medium (depends on caching algorithms)
Security – Perfect Forward Secrecy (PFS)	No	Yes	Yes
Security – Impersonation Risk	High (keys are shared, no detection of misuse, no common trusted infrastructure, depends on secure storage)	Medium (no common trusted infrastructure, depends on secure storage)	Low (depends on secure storage)
Constrained Environment	Optimized for Verification (benefits constrained servers)	Balanced	Optimized for Signing (benefits constrained clients)
Scalability (number of nodes interacting)	Low	Medium	High

Quantum computers[15] present new threats to existing cryptographic solutions because they are more effective at solving certain types of mathematical problems such as the integer factorization[16] problem, the discrete logarithmic problem,[17] or the elliptic curve discrete logarithm problem.[18] Current asymmetric cryptography algorithms reduce to one of these mathematical problems which are known to be solved by quantum computing more easily than traditional computers. Cryptographic algorithms are being designed that are thought to be secure against quantum computers are called *post-quantum* safe algorithms and has led to a new branch of cryptography study called post-quantum cryptography. Since asymmetric cryptography is most threatened by quantum computing, post-quantum asymmetric algorithm design is receiving a lot of attention currently. In contrast, symmetric key cryptography and hash functions are relatively secure against attacks using quantum computers. It is believed doubling the key size (e.g., from 128-bits to 256-bits) adequately protects against quantum computer attacks on symmetric algorithms.[19]

It is still too early to tell which quantum-safe algorithms will become an industry favorite for IoT given cost, power, and size constraints. However, it seems clear that where symmetric cryptography is already acceptable for IoT, it should continue to remain acceptable given a doubling of key size is the most economical quantum-safe solution. Quantum-safe asymmetric algorithms have much larger key size requirements or computation trade-offs, both of which apply to typical IoT platforms.

[15] https://en.wikipedia.org/wiki/Quantum_computing
[16] https://en.wikipedia.org/wiki/Integer_factorization
[17] https://en.wikipedia.org/wiki/Discrete_logarithm
[18] https://en.wikipedia.org/wiki/Elliptic-curve_cryptography
[19] Daniel J. Bernstein (2010-03-03). "Grover vs. McEliece" (PDF).

Summary IoT Framework Considerations

IoT frameworks came into being as a way to simplify development and deployment of IoT networks. The reality is IoT networks are inherently complex and, in many cases, necessarily so. IoT frameworks offer value because they create a data model abstraction that is simpler than applications having to deal with a myriad of message exchange options and dissimilar data definition. By allowing applications to focus only on the semantics of IoT node behavior and node interactions, interoperability improves. By hiding the complexity of connection establishment, bridging, gatewaying, and deployment of heterogeneous platforms, efficiency optimizations can be applied more uniformly. Although frameworks may increase complexity for simple deployment situations, they scale as deployments grow resulting in a simpler IoT system overall. Frameworks have other advantages, namely, they enable multiple views of the IoT system so manageability, resiliency, interoperability, security, safety, and usability perspectives can be represented. Complexity in any form, however, is a security consideration because vulnerabilities and security weaknesses can hide within the corners of complexity. Security practitioners should ask whether the framework is more complex than needed in order to realize the expected benefits, but also avoid workarounds that expose new attack surfaces.

IoT Framework Architecture

This section explores IoT framework layers in more detail. A following section looks at specific framework architectures that may be compared and contrasted. The majority of IoT framework architectures define three layers: (1) Data Object layer, (2) Node Interaction layer, and (3) Connectivity and Hardware Abstraction layer. This section also considers the Hardware layer, though it typically isn't considered part of an IoT

framework. However, because security necessarily should have ties to hardware, we've included a Hardware layer discussion. Security is integral to IoT framework layers revealing additional security insights relating to each layer (see Figure 2-6). This section explores each framework layer in detail with an emphasis on security.

Data Object Layer

The Data Object layer defines data structures that expose the "nodes" and their capabilities using a data definition language such as JSON.[20] One or more nodes may be hosted in the framework where one or more applications may interact with framework nodes via a framework API. Data objects are a set of attribute encapsulations. Some framework object models allow nested encapsulation with unlimited depth. Other frameworks limit nesting depth. The outermost encapsulation is the node. Since nodes logically correspond to an IoT network endpoint, it is given an identifier, NodeID, such as a Universally Unique Identifier (UUID) which is easy to generate dynamically given framework nodes may be transient. NodeID differs from DeviceID in that DeviceID is fixed in hardware. It is created during manufacturing and is used to facilitate device onboarding. NodeID typically is created in response to successful onboarding. Very constrained devices may use the DeviceID as the NodeID if the manufacturer has prevented the framework from supporting additional nodes.

The Data Object layer may define security objects such as access control lists (ACLs), credentials, and other device status information useful to management consoles and other nodes. Exposing security objects using the framework object model allows device and security management using the IoT framework infrastructure. The framework connectivity and interoperability properties make it a desirable ingredient for manageability. Security objects may expose values that are specific to a

[20] www.json.org

node such as credentials, ACLs, and NodeID or may expose values that are node independent or shared such as DeviceID, firmware, and hardware configuration.

The Security Objects in Figure 2-6 are useful for intranetwork and intradomain interactions. More sophisticated internetwork and interdomain interactions require an additional security layer that may be helpful for gateway operations. The gateway application contains control and management logic to present nodes to a peer IoT network that shadow actual nodes existing deeper inside the local IoT network. Gateway applications might even be used to bridge non-interoperable IoT frameworks. A following section explores interdomain security and framework gateways in more detail.

Node Interaction Layer

The Node Interaction layer contains messaging semantics and defines interfaces used for peer node interaction. Interface definition languages such as RAML and Swagger may be used to create machine- and human-readable interface definitions. A framework instance may support one or more messaging models, such as REST, publish-subscribe, and MESH. This layer ensures messages are formatted correctly, parses message contents, performs data consistency checks, and ensures messages are sent, queued, resent, or received properly.

Messages may require encryption and integrity protections. This layer maintains security associations between the local and peer nodes. Security associations identify the nodes, ACLs, privacy policies, and credentials (used to authenticate, authorize, and protect message contents). They may also define the security context from which to perform various security relevant operations such as encryption, decryption, signing, verifying signatures, enforcing access control, and so forth. The security context defines what is (or should be) the correct way to terminate the data exchange with peer nodes.

There are implementation challenges associated with security endpoint definition due to network layering. For example, a TLS or IP Security (IPSec) secured channel may be shared across multiple locally hosted nodes, implying nodes must use shared credentials, something frowned upon by most security practitioners. Alternatively, acceleration hardware may offload packet processing which may include offloading security operations too.

Ideally, the *Security Endpoint Context* is the central point of enforcement where the flow of data between the Data Object layer and the Connectivity layer can be inspected and controlled.

Platform Abstraction Layer

The Platform Abstraction layer defines the logical connection points available to framework nodes. Connection points support the messaging models available to the Node Interaction layer regardless of the capabilities of the underlying network stack. The Connectivity layer typically supports multiple connection points – one for each unique network stack. For example, the connection point, Conn-A, has a network stack consisting of HTTP, TCP, IP(v4 or v6), and Ethernet (Figure 2-6). A second connection point, Conn-B, has a network stack consisting of MQTT, TCP/UDP, IPv6, 6LoWPAN, and IEEE802.15.4. Connection points may be dynamically added or removed on more sophisticated platforms, while constrained platforms may embed a single connection point and network stack.

In some cases, the network stack includes message security technology such as IPSEC and TLS. The Connectivity layer depends on the Device Interaction layer for security associations specific to the node-to-node interaction semantics. This potentially divides the security enforcement point between the security side and the networking side. Some platforms are equipped with isolated execution technology that enables security processing within a network stack to be offloaded to a resource-isolated environment here referred to as a *container*. An alternative approach is

to move data protection into the Node Interaction layer. For example, OSCORE[21] defines a standard format for encrypting CoAP payloads before being given to the CoAP layer. This approach allows the security endpoint to move out of the protocol stack into the Node Interaction layer potentially simplifying implementation.

The basic idea is that the security endpoint context, data packets, and node-to-node security associations should exist within a suitably hardened container as a prerequisite for performing security relevant operations. Otherwise, there is opportunity for clever attackers to intercept, modify, view, or replace node objects.

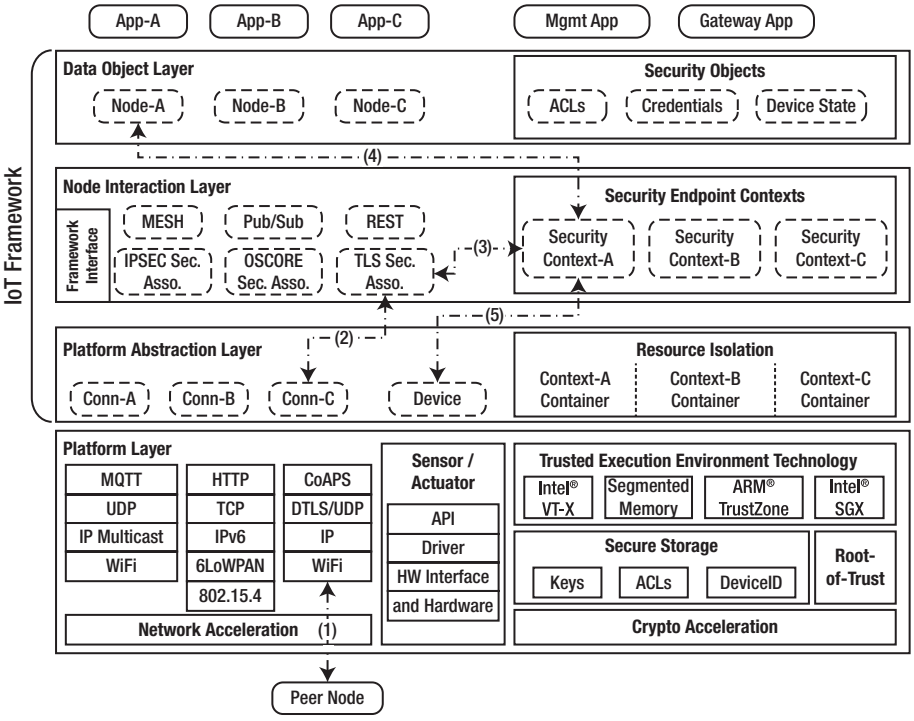

Figure 2-6. *IoT framework layers*

[21] https://core-wg.github.io/oscoap/draft-ietf-core-object-security.html

Platform Layer

The Platform layer beneath IoT frameworks can be divided into three categories: (1) networking, (2) sensor and actuator, and (3) security. The network layer focus is on efficient processing of network packets, quality of service, and power optimization. It also addresses network security threats related to malicious manipulation of network protocols. A common denial-of-service attack might flood the network with unexpectedly high volume of discovery packets. Discovery (aka *ping*) packets may not require prior authorization since the goal of discovery often implies finding out which credentials are most appropriate to use. Well-known attack mitigation techniques often are part of network hardware implementations, allowing the mitigation technique to be applied efficiently.

The sensor and actuator focus is on implementation of the main processing function of the IoT node which often represents the transition from IT to OT as native interactions are applied to the physical world. Otherwise, the node would just be manipulating data and couldn't be considered a *cyber-physical* system. The device driver and API are most often proprietary and specific to the vendor and model of the sensor or actuator. Vendor-specific behavior multiplied by the already large and expanding collection of IoT devices multiplies the complexities associated with multivendor interoperability. Hiding this complexity behind a common data object model is a primary reason for IoT frameworks.

The security focus is on hardening security-sensitive IoT functions. Trusted Execution Environment (TEE) technology isolates computing resources according to the various system tenants. IoT frameworks allow multiple tenants in the form of IoT nodes – nodes that may have different identities, security credentials, access policies, and configurations. Even in constrained environments where a single node is supported, there are security and device management scenarios that require tenant isolation for nodes performing administrative duties. The industry has a variety of TEE technologies that could be leveraged to harden IoT workloads that

include Intel SGX, Intel VT-x (virtualization technology), ARM TrustZone, and hardware memory managers that physically partition memory and other compute resources.

Secure storage is an essential element in IoT devices in that cryptographic keys, trust anchors, access control lists, and other policies need to be stored in ways that resist software attacks and ideally resist attackers who have physical access to the device. Replay protected memory is helpful toward preventing attacks on key exchange protocols, memory replacement, firmware update, and timing attacks.

Root-of-trust hardware is essential to the creation and protection of device identities that may be used to attest device security properties to a peer node and to security boot the device. Crypto acceleration hardware may offer additional protections as offloading encryption and signing operations may involve the use of a hardened coprocessor or ASIC (Application-Specific Integrated Circuit). Root-of-trust hardware or crypto offload hardware often includes a source of entropy necessary for generating encryption keys and trustworthy identifiers.

Security Challenges with IoT Frameworks

Security challenges are a reoccurring theme as we explore various IoT frameworks. Though they may have been designed with a wide range of security and privacy requirements, there are a few areas that are consistently problematic. IoT framework nodes are the logical endpoints in IoT networks, but the network layer context is often out of scope when operating at the framework Data Object layer (Figure 2-7).

In IP networks, endpoint nodes are identified by IP addresses, and routing logic is expressed in terms of IP addresses. Network layer identifiers are insufficient as IoT framework node identifiers. In IoT frameworks, nodes are logical and hence may share the same IP address but have different node identifiers. Linking encryption keys, authentication

credentials, and access control policies to IP address means security will not be granular enough and can't be consistently applied.

Uniform Resource Identifiers (URIs) and object identifiers such as Universally Unique Identifier (UUID) may be used to reference the framework's device nodes. For example, a URI might identify the framework context followed by an object identifier that is specific to the logical device instance – "href" : "oic://<Base64_encoded_UUID>/oic/d".

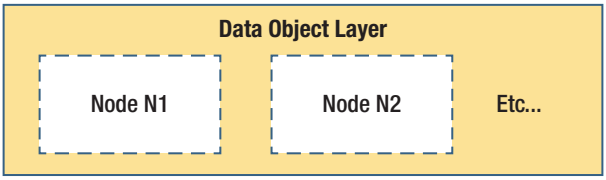

Figure 2-7. *IoT framework nodes are the logical endpoints in IoT networks*

The IoT framework node presents a security context where the security endpoint is an IP multicast address; using IPSEC implies the data protection ends at the network interface card or possibly inside a networking driver in an operating system. This leaves data exposed before it reaches the IoT framework's enforcement point where the decision is made to which node the data belongs.

A similar concern exists using Transport Layer Security (TLS) where data protections end within the operating system or within a network connection provider service. Connection services often expose APIs that a variety of applications may utilize. If the service isn't exclusive to the IoT framework, it is possible the cryptographic protections intended to terminate within the logical IoT device terminate within the service instead. Other applications serviced by the connection provider are at risk of becoming targets for attack because of the special access unwittingly given to them by the service. Care must be taken to ensure data carried over communication channels and messaging systems are protected by trusted execution environments that correspond to the expected logical device endpoint.

If data is protected using message-oriented techniques such as JSON Web Token (JWT), data protection may be extended into the IoT framework data abstraction layer, but there may be secure messaging library that is shared by all the logical device instances. A man-in-the-middle (MITM) attack could be successful if malware found a way to intercept the data after the data protection module is finished but before the logical device context is in place.

An IoT framework access path is depicted in Figure 2-6 where in step (1) a peer node accesses the IoT device through a Wi-Fi networking stack at connection Conn-C. In step (2) the Conn-C access path finds the TLS security association and the Security Context-A in the Security Endpoint Contexts. In step (3) access to decryption keys, ACLs, and role credentials is checked. The Security Context-A is a fulcrum point in the framework that uniformly applies an IoT network security policy involving the peer nodes and Node-A. Ideally, the security context operations are performed in a TEE that resists *man-in-the-box* attacks. If access is permitted, in step (4) the sensor/actuator hardware may be exposed to the peer node through Node-A data objects at step (5). Ideally, the entire access path will be isolated from the other nodes and operations occurring on the same device as the other tenants present security threats from within the device.

Consumer IoT Framework Standards

In this section, we explore several IoT framework architectures highlighting similarities and differences. In some cases, differences exist because different frameworks intend to address different requirements and use cases. In other cases, significant overlap of features and capabilities appears to exist because they address similar requirements but do so differently. This is unfortunate because it creates opportunities for incompatibilities. Such differences may be benign when used in isolated deployments but add significant complexity when interoperability across multiple deployments is desired.

Open Connectivity Foundation (OCF)

The Open Connectivity Foundation (OCF) was originally formed under the name Open Interconnect Consortium (OIC). Broadcom, Intel, and Samsung were among the initial founders of OIC and were later joined by Electrolux, Microsoft, and Qualcomm. IoTivity is the open source reference implementation of both OIC and OCF specifications. The OIC later became OCF when the AllSeen Alliance and OIC merged in October of 2016. The AllSeen Alliance is discussed in more detail in a following section.

The OCF framework (Figure 2-8) consists of three layers, Transports, Core Framework, and Profiles. The transport layer is a plugin interface that supports any number of transport plugin modules. Although the architecture refers to them as transports, the remaining networking layers (network, data link, and physical) are presumed to be provided as well. The OCF specifications do not prescribe how the layers are implemented, but the IoTivity reference implementation (see https://iotivity.org/downloads) may offer guidance. Support for various wired and wireless transports in IoTivity continues to grow. At the time of this writing, there was support for CoAP (UDP) over IPv4, IPv6, Ethernet, Wi-Fi, and Bluetooth LE. At the time of this writing, an Object Security for Constrained RESTful Environments (OSCORE) draft specification[22] defines a REST message binding to CoAP and HTTP. OSCORE supports connections originating in IoT networks based on a UDP transport that terminates in cloud services environments or remote access gateways that are based on a TCP transport.

OCF transport plugin module interface is transport agnostic, making it possible to define transport plugin modules that implement REST (Representational State Transfer) semantics. This implies OCF transport plugins could implement message queuing techniques such as MQTT (Message Queuing Telemetry Transport) or XMPP (eXtensible Messaging and Presence Protocol) without structural modifications to the framework.

[22] https://datatracker.ietf.org/doc/draft-ietf-core-object-security/

The transport interface interaction model roughly follows an object lifecycle pattern called CRUDN – Create, Retrieve, Update, Delete, and Notify. RESTful interaction semantics easily map to a series of request-response exchanges for each interaction – for example, Send Create_Request message followed by Receive Create_Response message. OCF interface semantics are typically defined using RAML[23] (RESTful API Modeling Language), although there is interest in migrating to Swagger[24] which complies with the OpenAPI specification. The OpenAPI specification[25] is an open source community effort aimed at defining robust data modeling languages and tools.

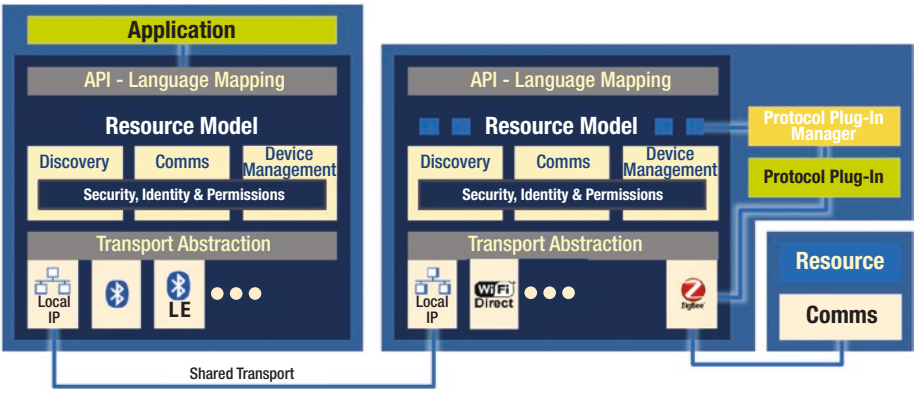

Figure 2-8. *OCF conceptual framework*

OCF Core Framework Layer

The Core Framework lies at the center of the OCF architecture. It defines the "resource" abstraction model which is arguably its most fundamental building block concept and the characteristic that most distinguishes it from other frameworks. An OCF *resource* is primarily a sequence of

[23] https://Raml.org
[24] https://swagger.io/
[25] https://github.com/OAI/OpenAPI-Specification

tag-value pairs but can have nested sequences as well. Resources are typically described using JSON. The OCF *resource model* approach to IoT networking presumes all aspects of the network can be represented declaratively, as a set of resource data structures having CRUDN interaction semantics. The traditional notion of a network topology consisting of nodes having routable IP addresses is hidden behind the resource abstraction.

Resources have several built-in properties (tags) that are common to all resources such as name, resource type, interface type, and whether or not it is discoverable and observable. Resource names are a URI (Universal Resource Identifier). Property names and name prefixes that are common to all are reserved by the OCF specification.

For example, "rt" refers to the resource type property, "if" refers to resource interface type property, "uri" is the resource name property if expressed as a URI, and "n" refers to a resource by its friendly name. Resource names prefixed with "/oic" are reserved for OCF use.

Additional properties may be appended that further specialize the resource. For example, it might define a property representing an operational state such as "on-off-state" where the accepted values are either ON or OFF. It might have another property "dim-level" with values in a range from 0 to 100, representing a light's brightness level.

This is a JSON schema representation of a simple resource:

```
"oic.r.switch.binary": {
    "type": "object",
        "properties": {
            "value": {
                "type": "boolean",
                "description": "Status of the switch"
            }
        }
}
```

This is a RAML representation of a CRUDN RETRIEVE interface definition:

```
get:
responses :
200: body:
          application/json:
             schema: |
                 { }
```

The point behind using interface and data modeling languages such as JSON and RAML is to enable the use of automated tools for generating code, tests, and even human-readable documentation that makes it easier to develop applications that not only interoperate but also can be adapted, updated, or modified at various operational stages.

The Core Framework layer defines several built-in resources used to implement several of the services and capabilities offered by the core layer. These include resource discovery, data transmission, data management, device management, security, identity, and permissions. Several built-in resources are listed in Table 2-2.

Table 2-2. *A Few Resources Built into an OCF Core Framework Layer*

Resource Name	Description	Functional Area
/oic/res	A resource that lists all discoverable resources known to the current network	Discovery
/oic/p	A resource that reveals details about the platform that hosts the OCF device	Discovery
/oic/rts	A resource that lists the resource type information for all discoverable resources	Discovery

(*continued*)

Table 2-2. (*continued*)

Resource Name	Description	Functional Area
/oic/ifs	A resource that lists the resource interface information for all discoverable resources	Discovery
/oic/mon	A resource that reveals observable resources	Device Management
/oic/sec/cred	A resource that lists the credentials this device has configured	Security Management
/oic/sec/acl2	A resource that lists the access control restrictions for this device	Security Management
/oic/sec/dots	A resource that facilitates device onboarding	Device and Security Management

A JSON representation of the /oic/p resource might appear as follows. Note this example includes comments denoted by double slash "//" which isn't defined by JSON:

```
/oic/p {
"rt": "oic.wk.p",
"if": ["oic.if.r"],
"pi": "ABCD123...",      //platform identifier UUID
"mnmn": "acme.org",    //platform manufacturer
"mnmo": "widget X",    //platform model number
"mnpv": "v1.0",        //platform version number
}
```

All properties of the /oic/p resource are read-only to support discovery use cases. A device management resource would likely allow update so a management console could configure the resource according to management goals.

The Core Framework specifications also define helpful building block resources that other resource designers may find useful such as *Links* and *Collections*. Links are a structure for defining a static connection between multiple resources. It consists of at least three parts: (1) the Context, (2) the Relationship, and (3) the Target and (4) additional parameters.

For example:

```
{
    "anchor": "/my/room/1",     //the Context
    "rel": "contains",          //the Relation
    "href": "/the/light/1",     //the Target
    "rt": "acme.light",         //the resource type
    "if": "oic.if.a"            //the interface type
}
```

The Collection resource is a bit like a Link resource only it contains an array of static connections to other resources.

For example:

```
/my/room/1 {
"rt": "acme.room",
"if": ["oic.if.r", "oic.if.rw"],
"color": "blue",
"dimension": "15bx15wx10h",
"links": [
    {"href":"/the/light/1", "rel":"contains", "rt":"acme.
     light",    "if":["oic.if.a", "oic.if.baseline"]},
    {"href":"/the/light/2", "rel":"contains", "rt"="mycorp.
     light", "if":["oic.if.s" , "oic.if.baseline"]},
    {"href":"/the/fan/1", "rel":"contains", "rt":"hiscorp.fan",
     "if":["oic.if.baseline"]}
    ]
}
```

OCF Profiles Framework Layer

OCF Profiles are libraries of resources containing common functionality (e.g., light bulb, pan-tilt-zoom camera). Profiles are grouped according to a target deployment context such as consumer, enterprise, industrial, auto, education, and health. Profiles are extensible. JSON validation ignores content not matching a schema target. OCF makes use of this behavior by allowing vendors to customize in any way they choose. We have mixed opinions regarding the use of this extensibility mechanism because, although it allows for post deployment customization, it also encourages the use of non-interoperable profiles.

The OCF data model supports resource introspection. Introspection can be used by a client to obtain a machine-readable description of all the resources, properties, and interface definition syntax. Introspection may be useful for systems that can learn how to interact with resources without prior programming.

The OCF Device Abstraction

OCF uses Universally Unique Identifiers (UUIDs) to identify OCF devices. The OCF device is like an OCF resource in that it has nested OCF defined Core and Profile resources. Core resources facilitate discovery, manageability, security, and connectivity. Profile resources define device type–specific data and behavior.

Access to OCF resources is accomplished using URIs. The OCF URI contains a device identifier in the form of a UUID followed by a reference to its resources. A client interacts with an OCF device by issuing a discovery message to identify available OCF server devices. This is followed by a RESTful message targeted at the device with interesting capabilities. The device's introspection resource may be used to gain additional insight regarding device capabilities and may be used to fine-tune subsequent interactions.

The OCF device abstraction logically defines a security boundary. OCF resource accesses follow CRUDN (Create, Retrieve, Update, Delete, Notify) interaction semantics that are part of the RESTful interface definition (e.g., PUT, GET, POST, DELETE). Access control policies use CRUDN privileges that are applied prior to returning resource data.

There can be multiple OCF devices hosted on the same physical platform. Logical devices are identified independently of the physical platform that hosts them. This means, from the perspective of the OCF device, it is not possible to distinguish whether a peer OCF device is geographically local or remote.

OCF Security

OCF security is exposed to devices through OCF resources. This is a simple yet powerful idea as all security interactions can be accomplished using the OCF framework. OCF security architecture has three main aspects: (1) access control, (2) message encryption, and (3) device lifecycle management. Access control is applied at the OCF device and resource-level granularity. It's worth noting that access control is not applied at the property level (although there are some exceptions). Access control list (ACL) policy is configured using the /oic/sec/acl2 resource. This resource is an array of ACL entries where each entry may be used to match the resource requestor to the requested resources so that an access restriction, expressed as CRUDN, can be applied before the requested resource is returned to the requestor.

```
/oic/sec/acl2 {
"aclist2": [
                "subject": ...,
                "resources": [...],
                "permission": CRUDN,
```

```
            "validity": ...,
            "aceid": INTEGER
        ]
}
```

The subject property is used to match the requestor. There are three ways this could be accomplished. One method uses the OCF device ID, which is a UUID. If the requesting device authenticates with a credential known to the local device, then the requesting device's ID is known. Another method is by role name. A role certificate may be presented at any time by the requestor during a session. If a role is asserted, then ACL entries that specify a role name could be used to match the requestor. A third method is by connection type. OCF connectivity options allow for anonymous (unauthenticated) and/or encrypted message payloads. It may be appropriate to supply a blanket ACL entry for anonymous requestors that is highly restrictive and only lessen restrictions when requestor is authenticated. Unencrypted data similarly may require a blanket ACL rule.

OCF supports a variety of cryptographic algorithms and key types including symmetric, raw asymmetric, and certified asymmetric. OCF devices must support symmetric keys and related algorithms. Security profiles may require support for raw asymmetric keys or keys with certificates.

Message encryption is applied by the transport layer (e.g., DTLS applied to CoAP messages). The use of TLS (Transport Layer Security) implies the endpoint where data is no longer protected by cryptography is somewhere in the framework but not necessarily in the OCF device context. The use of TLS also implies there are deployment cases where the TLS endpoint is actually a gateway, proxy, or firewall or another intermediate node that isn't the originating OCF device. Consequently, the use of TLS alone can't guarantee end-to-end data protection. To handle these, one of four options may be tried: the intermediary obtains

a copy of the OCF device's credential, the intermediary presents its own OCF credential (masking the true OCF device originating the request), the intermediary uses its own credential but supplies a role credential that is common to the originating device, or the intermediary remains anonymous.

While there may be several ways for an intermediary to establish a connection legitimately, the credentials used may not adequately enable the original requestor the appropriate access rights. Lack of end-to-end message protections can complicate management and deployment of proper security controls. Adding this complexity runs counter to the philosophy of simplifying apparent complexity while hiding actual complexity.

OCF has a device lifecycle management model that incorporates device lifecycle state into the device resource model. The /oic/sec/pstat resource includes a property named Device Onboarding State or "dos." There are five states:

- RESET: Device transitions to its default state prior to onboarding.
- RFOTM: Device transitions to a state ready for onboarding into a new network.
- RFPRO: Device transitions to a state ready for provisioning resources.
- RFNOP: Device transitions to a state suitable for normal operations.
- SRESET: Device transitions to a state subsequent to onboarding, but where the device may be recommissioned or reconfigured with other options normally established only at onboarding.

The device is guaranteed to be in one of these five states throughout its deployment. These states map to elements of an IoT platform lifecycle model (see Figure 2-5). For example, a device may be in the RESET state during manufacturing and supply chain phases then transition to RFOTM in order to enter the deployment phase. It may transition to RFPRO as part of onboarding and initial commissioning then transition to RFNOP while in normal operation and monitoring phase. Management and update phases may or may not require a transition to RFPRO depending on how impactful the changes may be to the framework's resources. Hardware or low-level system changes may require transitioning to SRESET in order to change resources and properties the framework expects are immutable. Decommissioning implies a transition to RESET.

OCF "dos" states can have beneficial security impact because the device model at the framework layer enforces restrictions that could otherwise be ignored (potentially resulting in security incidents) by other resources and applications. For example, the /oic/sec/dots contains a property "owned" that is only updatable when the device is onboarded into a network for the first time. It is read-only thereafter. If an attacker tries to update it in some way to force an ownership change, the device state model prevents it.

OCF onboarding accommodates secure supply chains. Owner Transfer Methods (OTMs) are secure protocols designed to work with platform embedded credentials such as a manufacturer's certificate. OTMs rely on participation from platform vendors to establish platform provenance at manufacturing and through the supply chain. A variety of OTMs are supported having various levels of provability of supply chain provenance. The OTM interface is extensible, allowing improved OTM adoption over time.

A security challenge facing OCF frameworks is the binding between the lower framework layer to the platform and its security capabilities isn't defined by the specification. Implementers are free to make trade-off decisions that likely differ from product to product and vendor to vendor.

The OCF resource model tolerates complexity in that it supports any data structure representable by JSON. OCF standardized structures, Links and Collections, can be used to create complex relationships between resources enabling, for example, unlimited layers of nested resources that are difficult to define meaningful ACL rules. Resources can contain links to resources hosted on remote devices resulting in a chain of interactions not bounded by an end-to-end ACL policy. Encryption is achieved using TLS. TLS endpoints occur in the communication layer resulting in hop-by-hop confidentiality protection semantics. Although the OCF resource model complexity may be justified, its flexibility shouldn't reach beyond the security mechanisms protecting it.

AllSeen Alliance/AllJoyn

The AllSeen Alliance began in 2013 as an open source Linux Foundation project that defined an IoT framework aimed at consumer class home and small office automation use cases. AllJoyn is the open source reference implementation that first became available in 2016. AllSeen Alliance member companies included Affinegy, Arçelik, Canary, Cisco, Changhong, doubleTwist, Electrolux, Fon, Haier, Harman, HTC, LIFX, Liteon, LG, Microsoft, Muzzley, Onbiron, Panasonic, Sears, Sharp, Silicon Image, Sproutling, Sony, TP-Link, Two Bulls, and Wilocity. The AllSeen Alliance merged with the Open Connectivity Foundation in October of 2016. IoTivity 1.3 released in June 2017 contained support for an IoTivity to AllJoyn bridge.[26] AllSeen deployments exist primarily as legacy networks as development resources have turned elsewhere.

AllJoyn architecture (Figure 2-9) consists of three classes of node, leaf nodes, router nodes, and bridges. Leaf nodes contain application code and are primarily responsible for authentication and encryption. Router nodes host leaf nodes – no direct application to application

[26] https://iotivity.org/downloads/iotivity-1.3.0

interaction is permitted unless brokered by a D-Bus (Desktop Bus) agent – though application nodes may embed router node functionality giving the impression of direct application connectivity. Router nodes are responsible for message exchange that includes request-response and publish-subscribe support. It handles discovery, advertising, presence, and session management. The messaging transport is provided by D-Bus[27] technology. D-Bus is a point-to-point communications protocol built on top of IPS (inter-process communication) or through TCP sockets. A daemon process monitors bus activity processing messages on behalf of its connected applications. D-Bus channels are named using UNIX filesystem objects. An application must know which transport protocol to use and an appropriate D-Bus name when attempting to connect to a peer leaf node known as the "bus address." D-Bus supports several status and discovery commands that may be helpful in determining the health of D-Bus daemon processes:

- Org.freedesktop.DBus.Peer is used to determine if a peer is alive.
- Org.freedesktop.DBus.Introspectable is used to obtain an XML description of the interfaces, methods, and signals the device implements.
- Org.freedesktop.DBus.Properties is used to expose native properties and attributes of connected devices or to simulate them if they don't exist.
- Org.freedesktop.DBus.ObjectManager is used to query subobjects under its path when device objects are organized hierarchically.

[27] https://cgit.freedesktop.org/dbus/dbus/tree/NEWS?h=dbus-1.12

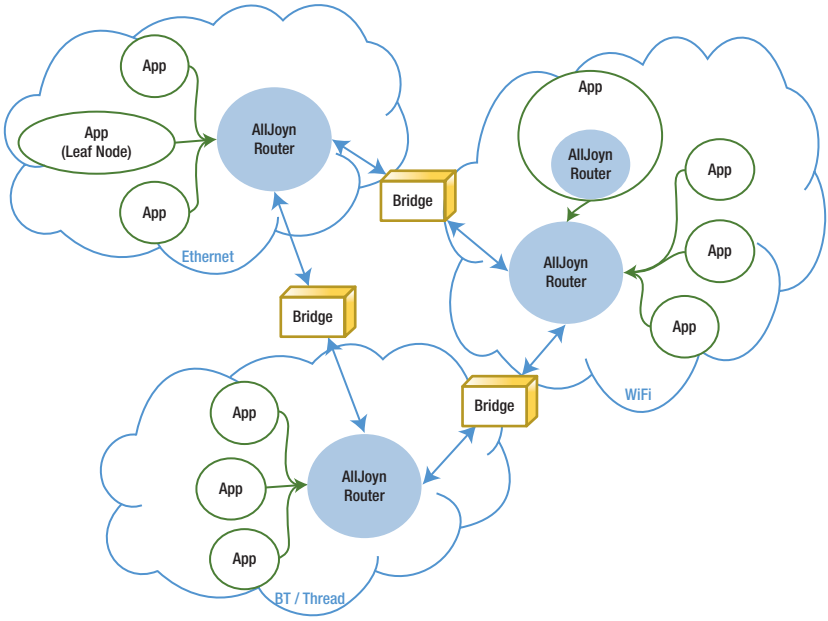

Figure 2-9. *Example AllJoyn network topology*

AllJoyn leaf node layers (Figure 2-10) consist of the AllJoyn Core that handles discovery, security, connection management, and network management. The AllJoyn Thin Client is an optimized subset of the AllJoyn Core targeting ultra-constrained environments. Message authentication and encryption protects the service framework and application data end-to-end. However, AllJoyn Thin Client nevertheless requires at least one routing node to complete an end-to-end connection.

The AllJoyn Service Framework implements device services. Onboarding, control panel, and notification services are common to all devices. Application-specific services are added as needed to expose device-specific specializations.

The AllJoyn router nodes contain an AllJoyn Core layer that contains message routing capabilities. However, routing nodes can be configured to protect all D-Bus traffic between cooperating D-Bus daemon processes using a common shared key. AllJoyn Management Functions perform advertising and discovery functions on behalf of leaf nodes. Routers

maintain context regarding leaf node presence and maintain a session for each attached leaf node. Messages involved in publish-subscribe messaging may have fan-out semantics requiring platform-level optimization support. For example, IP multicast may be an efficient way to deliver the same message to multiple recipients. Subscription registrations are maintained here as well. Message filtering can be applied by AllJoyn routers where the aim is congestion control given requests containing a query string.

AllJoyn bridges perform network and link layer translations when AllJoyn nodes are physically separated or when AllJoyn framework-level objects are gatewayed to a different IoT framework environment. For example, framework bridges may support IoTivity or OneM2M mappings.

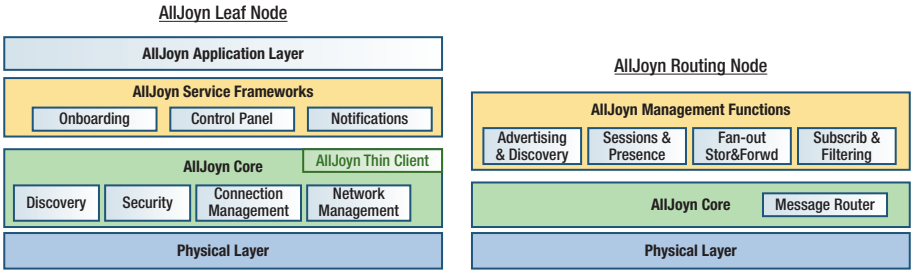

Figure 2-10. *AllJoyn leaf and router nodes layering*

AllJoyn Security

AllJoyn security rests with the AllJoyn leaf node and with the application layer. Such an approach encourages end-to-end protection of data. Effective data-level protection at the application layer requires data formatting and encapsulation technology that is part of its data model. AllJoyn data objects are described using XML and rely on XML Security[28] for secure encapsulation. Although D-Bus can support security at the IP layer, it relies on the application endpoint for end-to-end data protection.

[28] www.w3.org/standards/xml/security

When the Open Connectivity Foundation and the AllSeen Alliance merged, they defined a bridging specification that allows OCF and AllJoyn devices to interact; however the OCF bridging specifications do not define security interoperability.

Universal Plug and Play

Universal Plug and Play (UPnP) was originally designed for consumer electronics, mobile devices, home automation, and personal computer networks emphasizing *zero configuration networking* – the idea that setting up a service doesn't require any manual configuration. It includes automatic assignment of network addresses, automatic distribution of hostnames, and automatic discovery of network services. Although UPnP envisioned interoperation with consumer electronics and home automation, its first international specification published in 2008 by ISO/IEC[29] before the *Internet of Things* became a popular buzz word.

The UPnP set of standards has evolved to better support audio/video equipment, remote user interfaces, quality of service, and remote access from the Web. As recently as 2015, the UPnP Forum published the *UPnP Device Architecture 2.0*[30] specification that extends into the Web through XMPP integration. The *IoT Management and Control Architecture*[31] published September 10, 2013, addresses more directly home automation requirements with the inclusion of sensor management.

[29]"ISO/IEC standard on UPnP device architecture makes networking simple and easy." International Organization for Standardization. 10 December 2008. Retrieved 11 September 2014.

[30]www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf

[31]http://upnp.org/specs/iotmc/UPnP-iotmc-IoTManagementAndControl-Architecture-Overview-v1.pdf

The UPnP protocol stack (Figure 2-11) may be regarded as IoT frameworks, though loosely as UPnP is tightly bound to IP and the network services built around IP such as DHCP, DNS, IP multicast, and so on. UPnP network topologies parallel that of IP network topologies.

The UPnP Device Architecture layer consists of a discovery service named Simple Service Discovery Protocol (SSDP) that supports passive discovery request-response as well as active service availability notification and unsolicited advertisements using local multicast addressing. The General Event Notification Architecture (GENA) handles the details of registering notification events and sending notification messages when events are triggered. The Simple Object Access Protocol (SOAP) uses XML-formatted messages that are delivered using RESTful HTTP request-response exchanges. UPnP also supports IP multicast events for simple messages that need to be broadcast to multiple UPnP nodes.

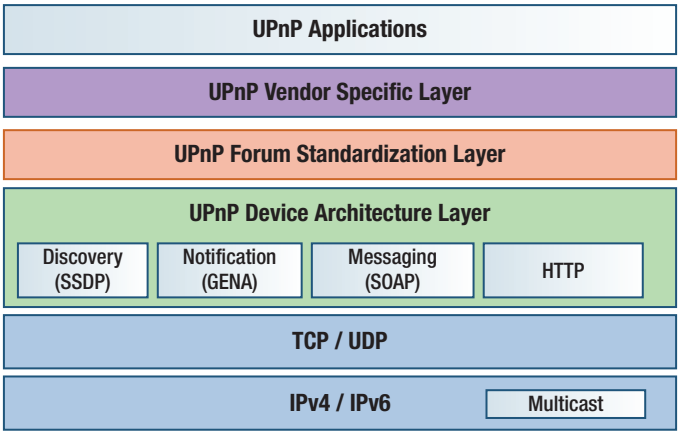

Figure 2-11. *UPnP protocol stack*

UPnP networks (Figure 2-12) consist of two node types, control points and Devices. Devices host Services. Device nesting is supported; the top-level Device is known as the Root Device. Devices are conceptual objects but are identified using IP addresses. Control points contain code that controls devices or otherwise interacts with services.

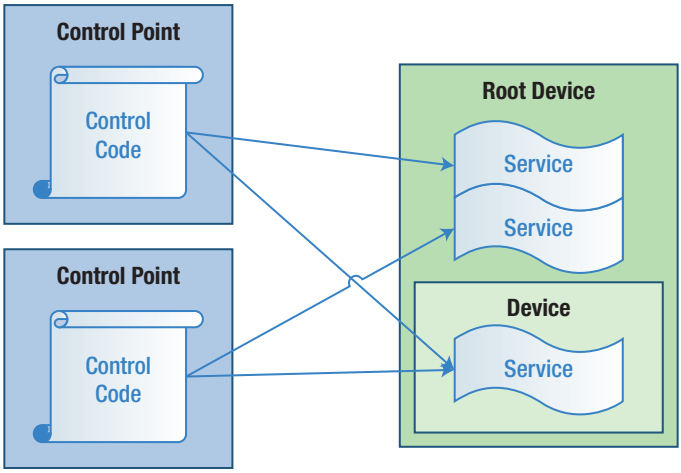

Figure 2-12. *UPnP network nodes consist of control points and Devices that host Services*

UPnP can be divided into six architectural elements: addressing, discovery, description, control, event notification, and presentation. Architectural elements roughly follow six phases of UPnP service and control point interactions:

I. **Addressing**: Zero-touch configuration motivated the use of DHCP (Dynamic Host Configuration Protocol) so the device would automatically look for a DHCP service to obtain an IP address. If no DHCP service was available, the UPnP device will autogenerate an IP address. The device can automatically obtain a DNS name using DNS forwarding. Secure device and control point identity was not a major focus.

II. **Discovery**: Service discovery automation is achieved through proactive "alive" messages that are broadcast periodically to listening control points. Control points can send discovery messages

with filters for the class of interesting service. This approach removes the need for statically configured services enabling dynamic services (that can go online or go offline easily). Control points rely on SSDP notifications to keep them appraised of service online status. Service name URLs are public which could have privacy implications. Secure discovery was not a major focus.

III. **Description**: Discovery reveals the existence of UPnP devices and services, but control points may require more context to determine if they are relevant to control point applications. Device description allows introspection using an XML description of the device structure. It includes the following information:

- Vendor-specific details include manufacturer name, model, version, serial number, and URLs to vendor-specific web sites.
- Service details include URLs for control, event notification, and service description. Service commands and their parameters are detailed.
- Variables that describe Runtime state are described in terms of data type, expected range, and event characteristics.

IV. **Control**: Control point code is expected to identify which commands and data objects are supported by the service to construct a program sequence that uses them to achieve application objectives. Command formatting is specified using SOAP protocol following the request-response pattern.

V. **Event Notification**: Services built around sensors and physical devices may change internal state autonomously. Control points seeking to be appraised of service and variable state changes can register for asynchronous notifications when things change. Notification messages are small; if the control point needs more information than is available in the notification message, it may need to follow the notification with a request-response interaction. UPnP event notification capability is referred to as the General Event Notification Architecture (GENA).

VI. **Presentation**: Normally, UPnP nodes operate as headless entities. Nevertheless, users may need to monitor and control things. UPnP services can support web browser user interfaces by returning a URL to a web page markup (HTML) that exposes service variables and control widgets.

UPnP Security

Initially, UPnP architecture did not comprehend security. It was thought to be addressed in the layers beneath (network) or above (application). More recently The IoT Management and Control Architecture[32] was added which included access control features for sensors was facilitated by roles and sensor permissions. Sensor permissions include

- ReadSensor: Control points can issue ReadSensor() actions to sensor objects.
- WriteSensor: Control points can issue WriteSensor() actions to sensor objects.

[32] http://upnp.org/specs/iotmc/UPnP-iotmc-IoTManagementAndControl-Architecture-Overview-v1.pdf

- ConnectSensor: Control points can issue ConnectSensor() and DisconnectSensor() actions to sensor objects.
- CommandSensor: Control points can modify IoTManagementAndControl properties in the data model (which is a data repository object).
- ViewSensor: Control points can read IoTManagementAndControl properties in the data model.

UPnP sensor objects expect control point operates with a particular role where permissions are assigned based on the set of behaviors each role is expected to follow.

UPnP control points must possess one of three UPnP defined roles:

- Admin: Role can read, write, connect, command, or view any sensor object.
- Public: Role can read or write specific sensor objects (e.g., those supporting the Public role).
- Basic: Role can read or write specific sensor objects (e.g., those supporting the Basic role).

A group of sensors form an object that can respond to control point accesses. Sensor groups have their own permission classification denoted by a sensor command name followed by the group name (e.g., smgt:ReadSensor()#[SensorGroupName]). There are four permissions for Read, Write, Command, and View. ConnectSensor isn't supported. Sensors inherit the group permissions upon joining the sensor group. Control points acquire the "group" access by joining the sensor group as a Control point. Interestingly the UPnP specification refers to group permissions as group roles.

UPnP security features are optional to implement, making it difficult to force the ecosystem to deploy UPnP with security.

The Open Connectivity Foundation and the UPnP Forum merged in 2016. They defined a bridging specification that allows OCF and UPnP devices to interact; however the OCF bridging specifications do not define security interoperability.

Lightweight Machine 2 Machine (LWM2M)

The Open Mobile Alliance (OMA) defined the Lightweight Machine 2 Machine (LWM2M)[33] specification to address IoT device management. We have included it at the end of the section summarizing consumer class IoT frameworks, but it could just as easily be classified as an IoT manageability framework. However, the Internet Protocol for Smart Objects (IPSO) Alliance extended LWM2M such that it can be used to describe a variety of consumer class IoT devices referred to as "smart objects" borrowing terminology from the LWM2M "object" model. OMA and IPSO Alliance merged in March 27, 2018,[34] forming new committees within OMA organization to continue its evolution as both an IoT manageability framework and a general-purpose IoT framework.

LWM2M Architecture

LWM2M architecture (Figure 2-13) utilizes a LWM2M Server node to host device management and other applications that interact with LWM2M client nodes hosting one or more LWM2M objects. Servers use RESTful CoAP commands (GET, POST, PUT, DELETE) to read and update the objects. Secure access is achieved using DTLS layer of CoAPs. CoAP and DTLS use UDP/IP and SMS transport protocols.

[33] www.openmobilealliance.org/release/LightweightM2M/

[34] www.omaspecworks.org/ipso-alliance-merges-with-open-mobile-alliance-to-form-oma-specworks/

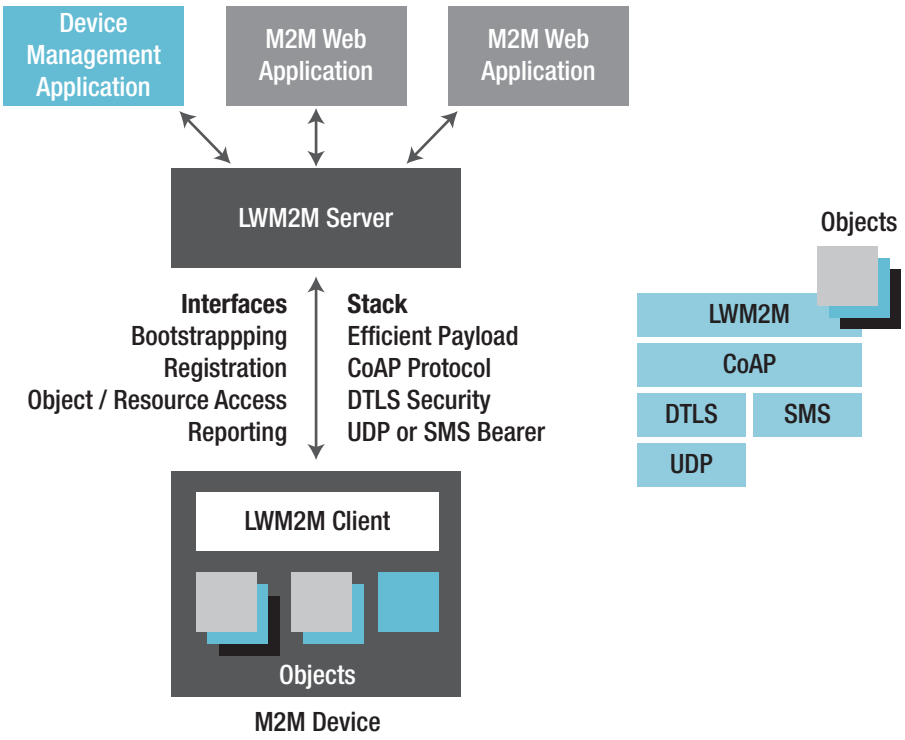

Figure 2-13. *LWM2M architecture showing client node with objects being managed by a Server node hosting device management and various web applications*

The LWM2M object model (Figure 2-14) is a simple but powerful abstraction of IoT devices. The LWM2M client is the managed node and corresponds to a sensor/actuator device. LWM2M nodes describe a set of network exposed variables called objects. A LWM2M Server may reference an object using a URI string that names the object plus its resources. For example, a LWM2M URI might appear as "/0/1" where "0" is the object identifier and "1" is the resource identifier. Objects contain one or more resources, but resources may not contain objects; in other words, nesting of objects is not supported. Friendly names are not supported since doing

so was thought to make URIs unnecessarily verbose. Instead objects and resources are numeric values. It is possible to have an array of objects of the same type using same object identifier. An Object Instance Identifier is added between the object ID and the resource ID to qualify the object instance. The URI format has the following form:

```
/ <ObjectID> / <ObjectInstanceID> /
<ResourceID>
```

Figure 2-14 shows an example object configuration consisting of two objects. The first contains a single object instance with three resources. The URI path begins with a leading slash "/" followed by the ObjectID referencing the first object (denoted by red arrow). It is followed by a second slash then the ResourceID referencing the third resource in the first object (denoted by a green arrow). The second object contains two instances of Object 2 where each instance consists of six resources. The URI path examples have three elements, the middle being the Object Instance Identifier (denoted by a blue arrow). One URI path shows an Object Instance Identifier with the value 1 that references the first object instance and the first resource instance within it. The other URI path references the second object instance and the sixth resource within it.

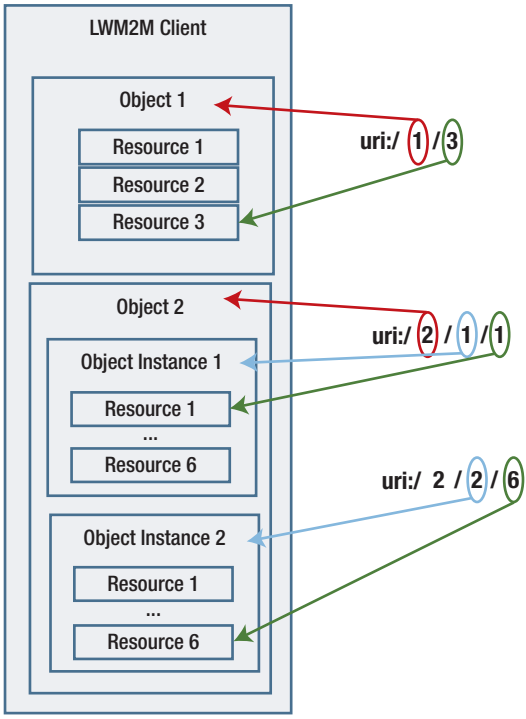

Figure 2-14. *LWM2M object model example showing URI references to data values*

The LWM2M object model expects IoT devices can be described relatively simply. The object model abstraction may hide significant actual complexity requiring the object model designer to think carefully about which device attributes need to be exposed and how best to map actual complexity to a simpler apparent complexity.

The example object in Figure 2-15 reveals six resources. The chart describes additional metadata regarding the resource including the type of access allowed (read vs. read/update), if it is a multi-instance object, the resource data type, the allowable range of data values, and the units in which the data is expressed.

Resource Name	ID	Access Type	Multiple Instances?	Type	Range	Units	Descriptions
Latitude	0	R	NO	Decimal		Deg	The decimal notation of latitude, e.g. -43.5723 [World Geodetic System 1984]
Longitude	1	R	NO	Decimal		Deg	The decimal notation of longitude, e.g. 153.21760 [World Geodetic System 1984]
Altitude	2	R	NO	Decimal		m	The decimal notation of Altitude in meters above sea level.
Uncertainty	3	R	NO	Decimal		m	The accuracy of the position in meters.
Velocity	4	R	NO	Refers to 3GPP GAD specs		Refers to 3GPP GAD specs	The velocity of the device as defined in 3GPP 23.032 GAD specification. This set of values may not be available if the device is static.
Timestamp	5	R	NO	Time			The timestamp of when the location measurement was performed.

Figure 2-15. *Example LWM2M location object*

The object namespace needs to be managed to avoid confusion when servers access client objects. The OMA reserved object identifiers 0–1023 for OMA defined objects. 1024–2047 are reserved for future use. 2048–10240 are allocated for third-party defined objects. For example, the IPSO Alliance object definitions are allocated from this range. 10241–32768 are assigned to public entities, vendors, or individuals for proprietary use.

Introspection is not supported except through the use of a separately defined introspection service – something that wasn't defined at the time of this writing.

LWM2M Device Management

LWM2M defines five device management services:

- Bootstrapping: Configures symmetric secrets, raw public keys, and certificates clients and service will use to establish DTLS sessions. LWM2M Services may be configured. Access control lists may also be configured.
- Remote Management: Updates operational settings as defined by device profiles. Triggers for controlling actuation may also be configured or reset as part of normal operation.

- Firmware Update: Client nodes report firmware version and firmware packages can be installed through the firmware update object.
- Fault Management: Device errors can be exposed through the fault reporting objects. These may be viewed by other nodes querying operational status.
- Reporting: Notification of changing sensor values can be configured for multiple recipients. Status of the notification can be monitored and configuration changes applied when needed.

The LWM2M architecture model reverses client and server roles (Figure 2-16) in comparison to other frameworks such as OCF, UPnP, and AllJoyn. This seems reasonable since the primary goal of LWM2M is device management where the device utilizes management service providers that bootstrap and configure the client. LWM2M supports both client- and server-initiated bootstrapping. Once the client device is configured, it may interact with other IoT nodes as an IoT service such as a sensor or actuator.

It may be reasonable to combine LWM2M for device management with a different IoT framework that doesn't support device management since LWM2M can operate alongside it provided the other IoT framework device lifecycle states are aligned with the LWM2M device state model.

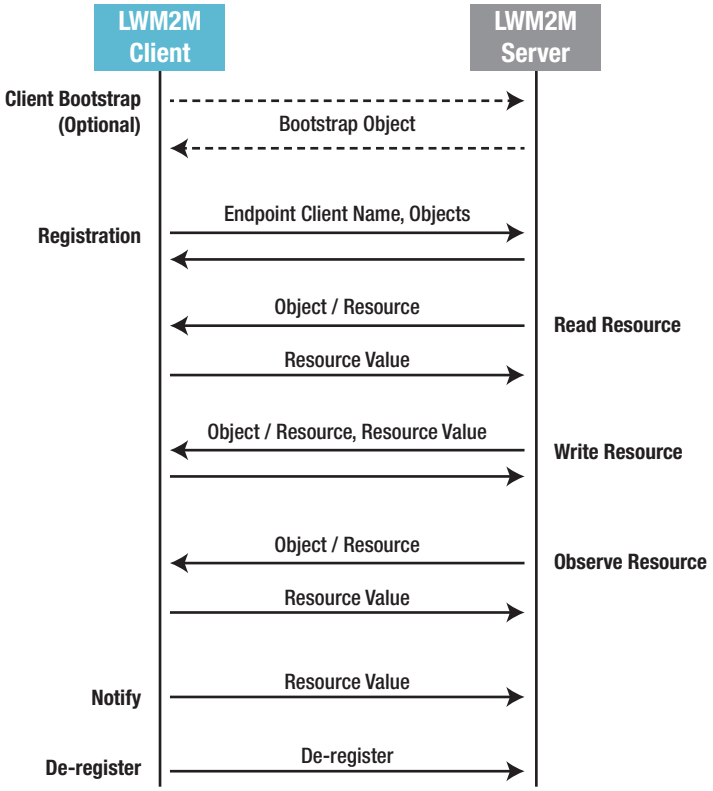

Figure 2-16. *LWM2M example device management lifecycle scenario*

LWM2M Security

LWM2M security has two main components, DTLS secured messages and access control lists (ACLs) for LWM2M objects (Figure 2-17). DTLS supports shared secrets (symmetric) using cipher suites for preshared keys (PSK), raw public keys (asymmetric) using cipher suites that perform ephemeral Diffie-Hellman key exchange that supports perfect forward secrecy (PFS), and certificates (asymmetric) using cipher suites that support popular certificate signing algorithms such as elliptic curve cryptography and RSA.

ACL support is achieved using the Bootstrap server to provision access control resources to LWM2M clients seeking access to LWM2M servers. In the following example, the Bootstrap server provisions the security object in Client 1 with the ACL object with read and write access to Server 1 (e.g., ACL:<Server 1, RW>). It also provisions Client 3 with read and write access to Server 2 (e.g., ACL:<Server 2, RW>).

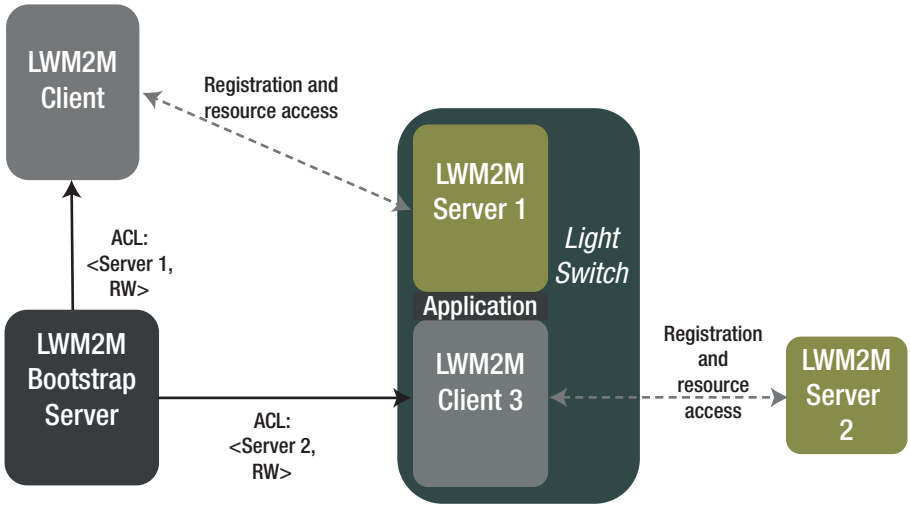

Figure 2-17. *LWM2M access control list configuration*

Provisioning credentials to each of the clients to allow the Bootstrap server access to their security objects is part of initial device setup, but LWM2M doesn't (at the time of this writing) implement onboarding (see the section "Deployment"). The method for establishing trust in the Bootstrap server by devices is vendor specific.

One Machine to Machine (OneM2M)

Eight global standards organizations [ARIB (Japan), ATIS (United States), CCSA (China), ETSI (Europe), TIA (United States), TSDSI (India), TTA (Korea), and TTC (Japan)] and six other industry fora, consortia, or standards bodies (Broadband Forum, CEN, CENELEC, GlobalPlatform, Next Generation M2M Consortium, OMA) collaborated to develop the OneM2M standard. The group, known as OneM2M,[35] was formed in July 2012. OneM2M produced the OneM2M technical specification in February 2016.[36]

OneM2M is an architecturally complete IoT framework (Figure 2-18) that consists of three basic layers: (1) Application layer, (2) Common Services layer, and (3) Network Services layer. An instantiation of a layered module is called an entity. An application is therefore an *application entity (AE)*, a service is a *common services entity (CSE),* and a network module is a *network services entity (NSE)*. Interfaces facilitate communication between entities known as *Reference Points*. A OneM2M reference point uses the nomenclature "Mc-" meaning M2M communication to the entity "-" – where the dash is a placeholder for the first letter of the entity name. For example, Mca describes a reference point connecting an Application Entity and a Common Services Entity. Mcn describes a reference point connecting a Network Services Entity to a CSE. Mcc describes a CSE to CSE reference point.

[35] http://onem2m.org/

[36] OneM2M Technical Specification, TS-0001-V1.13.1, Functional Architecture, 2016- February-29

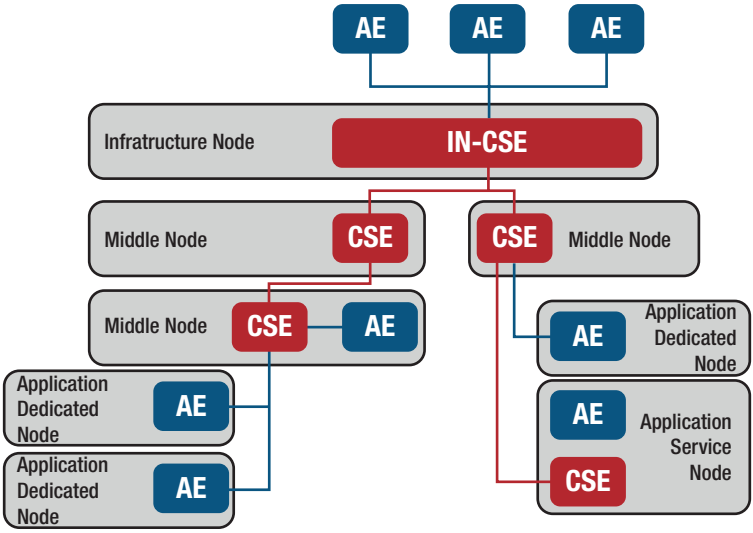

Figure 2-18. *OneM2M node topology architecture*

Deployment scenarios may have stereotyped nodes, according to a logical or functional network topology. For example, Application and Common Services entities may cooperate to provide infrastructure capabilities such as manageability services, message logging, telemetry, and so on. OneM2M refers to these nodes as infrastructure nodes (IN). Other nodes may cooperate to implement an application, for example, HVAC control, called Application Dedicated Node (ADN) or Application Service Node (ASN). Nodes deployed to connect ADNs to INs or other ADNs are called middle nodes (MN). Bridging non-OneM2M nodes are given the acronym NoDN.

Nodes may contain programs that control resources on other nodes. Resources are composed of a set of attributes. Resources can be nested, called a *child* resource.

Nodes are identified with a globally unique identifier that is assigned when the node registers with a registration node hosting a registration common services function. Physical devices host OneM2M nodes.

Entity layers are subdivided into *functions*. The Common Services Entity (Figure 2-19) defines a handful of common services functions (CSF) that manage device lifecycle including the following:

- Application and service layer management (ASM): The ASM function manages all entities hosted by any node excluding NoDN nodes. Management functions consist of two categories: (1) configuration functions and (2) software management functions. Configuration CRUDN functions expose resources used to manage entities, while software management functions are concerned with managing software and related artifacts associated with a software lifecycle.
- Communication management and (message) delivery handling: These functions manage delivery, temporary storage, and caching of messages. It also manages policies related to configuration and tuning of message delivery infrastructure.
- Data management and repository handling: These functions manage data repositories. They are concerned with the collection, aggregation, mediation, storage, and preparation for analytics and semantic processing.
- Device management: These functions address device management capabilities associated with OneM2M nodes and can use existing IoT device management frameworks such as TR-069 and LWM2M or may define new functions. Device management functions translate data, protocol, and semantics from one management node to another using a *Management adapter* module. Management gateways, proxies, and bridging functions

fall within the scope of device management functions. Device management functions perform device configuration, device diagnostics, monitoring, firmware management, and topology management.

- Discovery: Nodes, resources, and attributes can be discovered using a discovery CSF. Typically, the invoker supplies a query value that selects a subset of available possible matches. Filter criteria are expressed in terms of identifiers, keywords, location, and other semantic information.
- Group management: Nodes can be organized into groups. The group management CSF must validate group membership and whether the group member is capable of performing functions meaningful to the group. Groups are used to coordinate publication, broadcasts, or multicasts to multiple nodes and to define roles for access control.
- Location: The location CSF senses and publishes location information for the node. Location coordinates can be more than latitude-longitude coordinates but require knowledge of location extension semantics.
- Network service exposure: The network service exposure, service execution, and triggering (NSSE) CSF manages exposure of underlying networks and communication layers through Mcn reference points and NSE modules.

- Registration: Entity services must register with a registrar CSF in order to make their services available for use. The registration CSF supplies a requestor with the node identifier where the service can be reached, a schedule for when it can be reached, and details for accessing the service.
- Security: The security CSF handles identity management, access control, authorization, authentication, security associations, data confidentiality, data integrity, and security system management. Access control list subjects can group nodes that enforce read or write permissions. ACLs are associated with resources, entities, and repositories. Access control can be applied to discovery resources but requires subject authentication and authorization – though an "anonymous" group could be defined that corresponds to an ACL entry matching unauthenticated subjects.
- Service charging and accounting: The SCA CSF manages telemetry generation and collection used to charge for services, events, information, and real-time credit control.
- Subscription and notification: The subscription CSF manages subscription operations and notification message delivery to subscribers when the subscription condition is met. Subscriptions are registered with a resource or group of resources following an access control check. Changes to resources are tracked at attribute granularity. Changes to subresources are also tracked but not attributes of subresources.

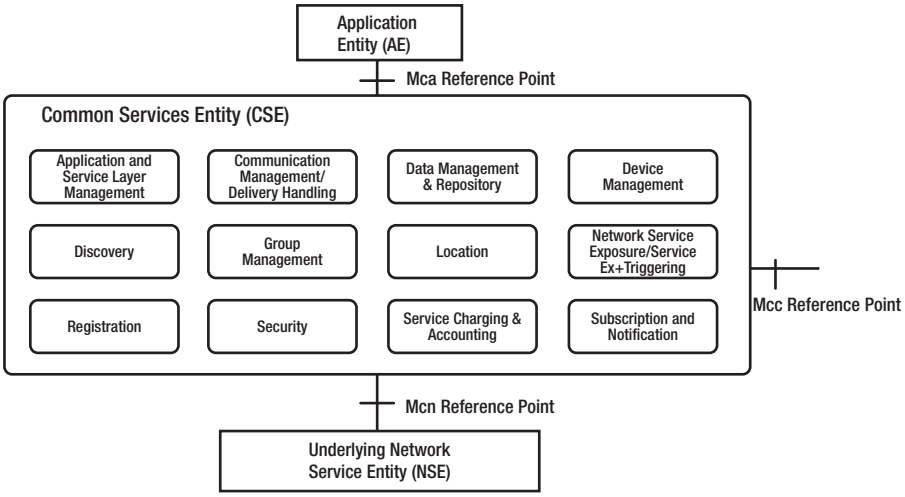

Figure 2-19. *OneM2M layering with entities and Common Services Entity functional modules*

IoT networks are sometimes partitioned into enclaves of subnetworks called *domains* (Figure 2-20) to improve isolation for safety, reliability, and security reasons. OneM2M reference point architecture envisages network enclaves by allowing multiple AE + CSE + NSE verticals connected through peer Mcc and Mca reference points. For example, a fieldbus domain may contain a network of closed-loop sensors and actuators running at real time or near real time, while an infrastructure domain may contain accounting, telemetry, firmware update, and other services based on restful client-server interactions. Still another domain may offload complex analytics to a data center or Cloud.

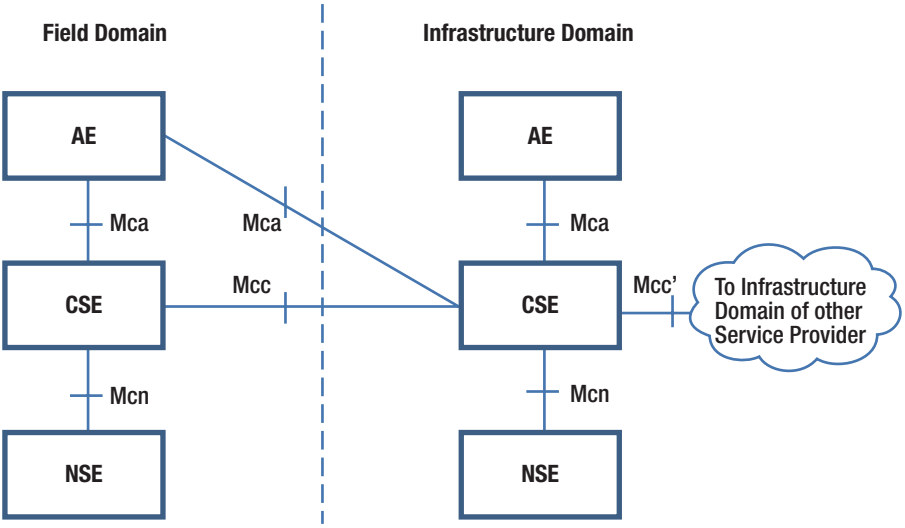

Figure 2-20. *OneM2M domain architecture allows network enclave isolation*

OneM2M device management is built from an open-ended set of common services functions that may be tailored toward any number of existing industry standard and nonstandard device management solutions including TR-069,[37] OMA-DM,[38] and LWM2M. As such, OneM2M can be thought of as a framework of frameworks.

OneM2M architecture allows extremely flexible configuration of functional modules and extensibility options. This flexibility may be helpful when tailoring a solution for constrained embedded devices seeking to minimize resource footprint or when designing gateways, bridges, and framework service nodes that are scattered throughout a complex IoT network. However, flexibility may come with a cost as

[37]Broadband Forum Technical Report, "TR-069 CPE WAN Management Protocol," Issue: 1 Amendment 6, Version 1.4, March 2018. www.broadband-forum.org/technical/download/TR-069.pdf

[38]www.openmobilealliance.org/wp/overviews/dm_overview.html

network latencies, routing, network security, and network management overhead may be incurred. Hiding this complexity from system designers may have undesirable consequences, while exposing the flexibility (having simplified apparent complexity) to applications and users may also have undesirable consequences.

OneM2M Security

OneM2M security design comprehends identity, authentication, authorization, access control, data protection, and privacy. That is to say, each of these security requirements was considered and addressed to a certain extent. However, the test determining adequacy largely depends on how completely the industry implements the standard and how effective the security mechanisms defined address the threats facing IoT networks.

OneM2M security administration begins with the provisioning of *master credentials* that enables the security CSF functions to be applied. Master credentials can be post-provisioned (subsequent to initial deployment of a CSE containing security CSFs) or pre-provisioned with cooperation from a device manufacturer – though the exact operation of onboarding protocols for pre-provisioning is out of scope.

OneM2M framework architecture abstracts away (hides) physical (device) boundaries. An Mcc reference point may or may not cross a device boundary. The same is true for Mca reference points as well. Intuitively, one might conclude that the use of an Mcn reference point does cross a physical boundary, but with IP loopback, shared memory, and other interprocess communication and overlay network mechanisms, Mcn also doesn't describe physical boundary crossing semantics. This is relevant to security because attack points often occur at boundary crossings. Although the specification intends security CSF functionality will "protect" security-sensitive information, there are a wide variety of hardware and software mechanisms to draw from – each having differing security and privacy properties.

Industrial IoT Framework Standards

The IoT framework standards discussed up to this point primarily address consumer grade IoT applications and deployments. That doesn't mean the standards organizations and member companies could not extend their architectures to accommodate requirements typically associated with industrial IoT. This section considers IoT frameworks that were designed specifically to address industrial control system requirements. Industrial control systems predate the Internet of Things and even predate the Internet. Fieldbus technology is the foundation of process automation, building automation, and automated manufacturing. This section doesn't survey the vast expanse of "brownfield" fieldbus technology.[39] Instead, it focuses on Industrial IoT (IIoT) standards that aim to improve interoperability through appropriate use of inexpensive, ubiquitous Internet technologies and are supported by a rich ecosystem.

Industrial Internet Control Systems (or just Industrial Internet Systems – IIS) may be a more appropriate terminology than IoT because at their core are complex semiautonomous and fully autonomous process automation systems that operate at a level of sophistication that clearly goes beyond consumer IoT. They pay close attention to Quality of Service (QoS), Quality of Experience (QoE), and safety requirements.

The architectural principles defined by the IIC reference architecture serves as a reference point for evaluating the merits and demerits of IIS framework solutions. The next section highlights important elements of industrial IoT system architecture as defined by the Industrial Internet of Things Consortium (IIC). In subsequent sections, we also highlight the Open Platform Communications-Unified Architecture (OPC-UA) and Data Distribution Services (DDS) open source IIS frameworks.

[39]<tbd Reference to industrial control systems>

Industrial Internet of Things Consortium (IIC) and OpenFog Consortium

The Industrial Internet of Things Consortium (IIC) was formed by AT&T, Cisco, IBM, Intel, and General Electric in November of 2016. The IIC created a reference architecture[40] for IIS that considers common needs and challenges pertaining to control systems in energy, healthcare, manufacturing, public sector, transportation, and factory automation.

In December 2018, the IIC and OpenFog Consortium agreed to join forces under the name IIC.[41] The OpenFog Consortium was founded by ARM Holdings, Cisco, Dell, Intel, Princeton University, and Microsoft in 2015. OpenFog Consortium and IIC both focused heavily on industrial IoT architecture.

Industrial Internet Systems bring new levels of performance, scalability, interoperability, reliability, assurance, and efficiency to the forefront. As such, the IIC determined it should produce a reference architecture first (and not an IoT framework[42] and a reference implementation). IIS systems often operate in mission critical environments that require real-time or near real-time responses and are "smart" through increased integration with higher-level networks that include enterprise resource planning, information technology administration, analytics, and big data correlation engines.

One aspect of the IIC architecture helps us understand the implications of transforming the largely isolated brownfield embedded control systems and technology into something that benefits from

[40]The Industrial Internet of Things Volume G1: Reference Architecture IIC:PUB:g1:V1.80:20170131 https://www.iiconsortium.org/IIC_PUB_G1_V1.80_2017-01-31.pdf

[41]www.smartindustry.com/industrynews/2018/iic-and-openfog-consortium-join-forces/

[42]Note to reader: The IIC specification refers to sub-architecture sections as "frameworks" not to be confused with our usage.

the Internet economies of scale and its robust ecosystem. Industrial embedded control systems have existed before the popular Internet and have evolved alongside it for several years. Its evolution into the IIoT seems inevitable, but doing so creates a complex problem for interoperability given the existing brownfield systems will likely continue for many years.

It is not our objective to deeply explore the IIC reference architecture here. However, the reader might appreciate the role of a reference architecture when evaluating IoT frameworks as building blocks of IIS systems. Different parts of an IIS ecosystem bring different viewpoints (Figure 2-21) of the system. The IIC reference architecture explores IIS from four viewpoints:

- Business viewpoint: Identifies stakeholders, business objectives, values, vision, and related regulatory context and comprehends business-oriented concerns.
- Usage viewpoint: Represents the activities, sequences, and functionality involving human or logical users. It ultimately establishes whether the IIS achieves value from the user's perspective.
- Functional viewpoint: Identifies functional components, structures, interfaces, interactions, and relationships. It considers trade-offs associated with the interests of systems architects, component architects, developers, and integrators.
- Implementation viewpoint: Considers challenges and implications of functional components, their communication, and lifecycle procedures and dependencies.

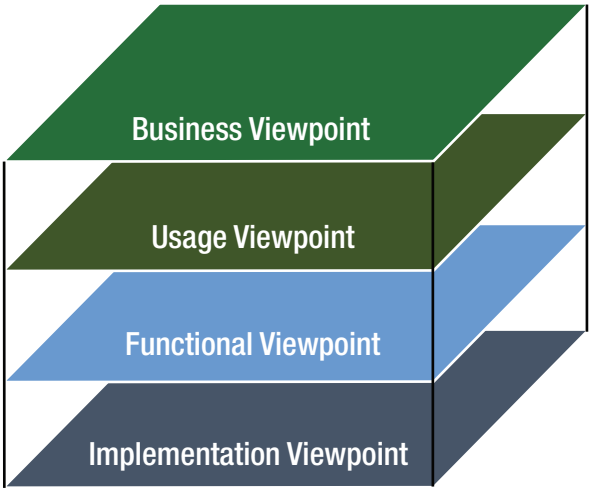

Figure 2-21. *IIC reference viewpoints*

Although multiple viewpoints exist, security objectives can be frustrated if a perspective somehow becomes hidden from the others in the context of continuous security monitoring, threat detection, decision making, and response management. For example, security return on investment value may be weighed against performance or consumer satisfaction value. The user benefits of autonomous operation (without users) may be compared to perceived and actual benefits of user involvement in setting and evaluating security relevant decisions. Security functional viewpoint defines points where security-related enforcement and decision making may impact other functional goals. The implementation viewpoint applies security technologies involving patterns and system components in ways that are correctly implemented and easy to maintain and ensure correct operation of security functions, algorithms, and hardening.

The IIC functional viewpoint reference architecture (Figure 2-22) recognizes an important understanding of IIS systems having five functional domains that must coexist as interoperable subsystems while ensuring appropriate isolation mechanisms prevent the goals of each domain from being compromised given failure or compromise in a peer domain.

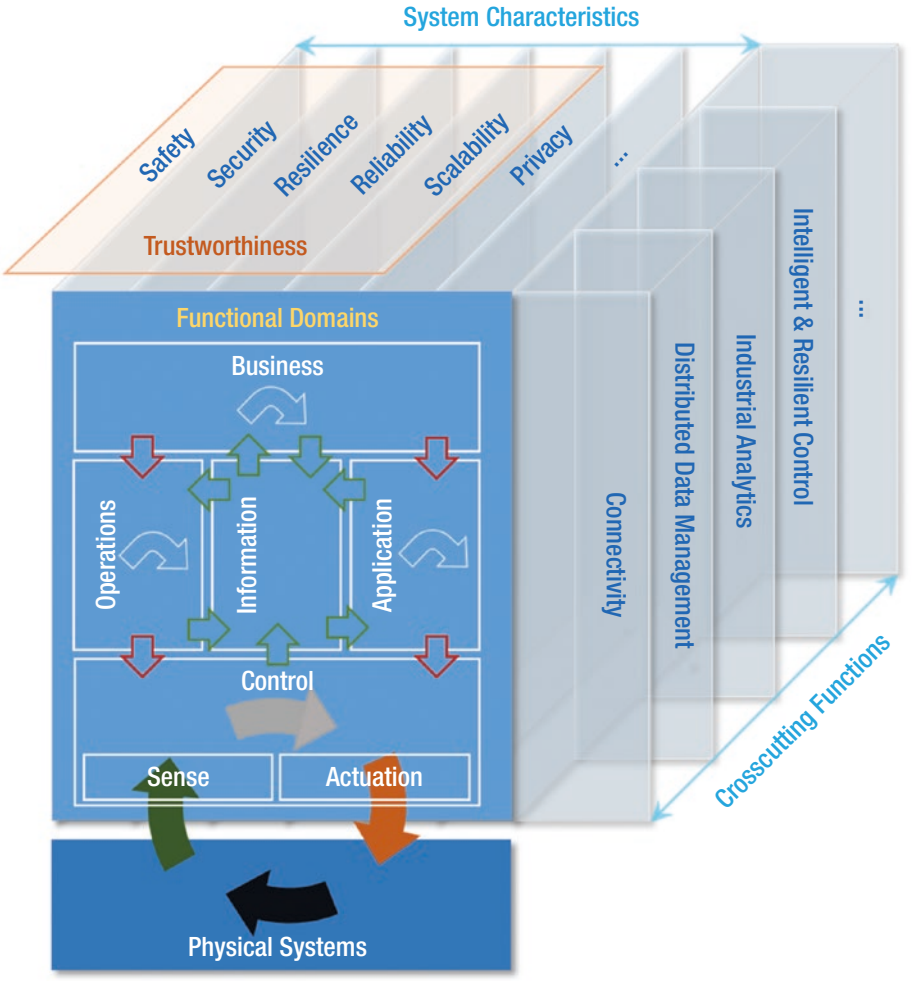

Figure 2-22. *IIC functional viewpoint reference architecture showing various functional domains*

The business domain functions as a layer on top of operations, information, and application domains that interact with the control domain. The control domain consists of a separation between cyber and physical systems brokered by sensing and actuation functions. User interactions may occur at each domain according to domain-specific

Quality of Experience objectives. Cross-domain interactions should carry the appropriate level of domain-specific context to ensure peer domain functions do not, in some way, misinterpret the semantics of command interaction, control flow, and data representation as this can result in failures and security vulnerabilities.

The IIC implementation viewpoint reference architecture (Figure 2-23) captures an important three-tier network topology structure that recognizes an Edge Tier network consisting of sensor, actuator, and controller nodes that may share latency, resiliency, and QoS requirements that typically are met by Edge-class technologies. These differ from Platform Tier technologies used to implement scalable, reliable, available systems for data analytics, operations, and data transformation. Similarly, the Enterprise Tier consists of technologies tuned for system maintenance, management, and system-level controls. Inter-Tier interactions are held in check through bridging, gatewaying, and proxying technologies aimed at preserving the correct context of the peer Tier when performing control operations or when moving data between Tiers.

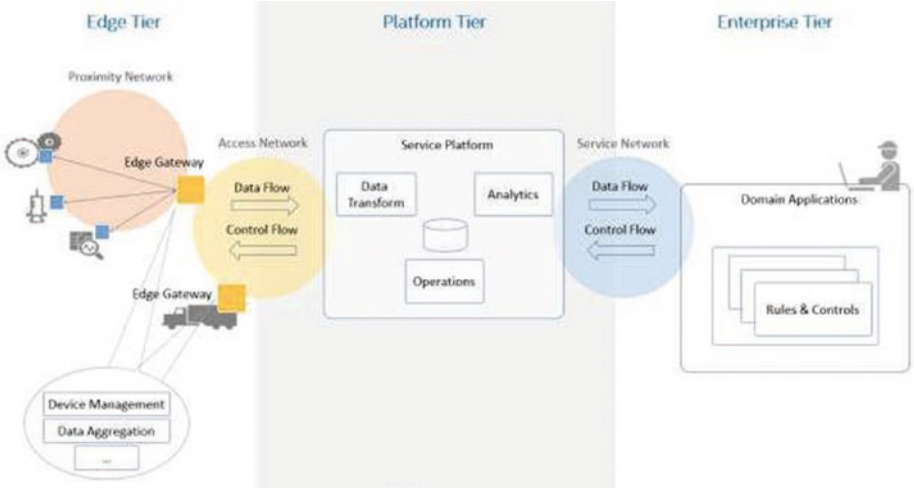

Figure 2-23. *IIC implementation viewpoint reference architecture showing a three-tier network*

Multiple viewpoints can be combined to reveal additional insights regarding an IIS system. For example, Figure 2-24 shows the functional viewpoint architecture overlaid with the implementation viewpoint architecture. The Control Domain exists in the Edge Tier which contains the Proximity Network consisting of sensors, actuators, controllers, and gateways to Platform Tier. The Information and Operations Domains exist in the Platform Tier bridging the Access and the Service Networks. The Platform Tier contains data service and platform management, data distribution, persistence, streaming, aggregation, and transformation. The Operations Domain is concerned with provisioning, deployment, metadata, monitoring, telemetry, optimization, and access control. The Application and Business Domains exist in the Enterprise Tier extending the Service Network with business analytics, CRM, DSS, BSS, and so on and enterprise applications, APIs, portals, and enterprise rules.

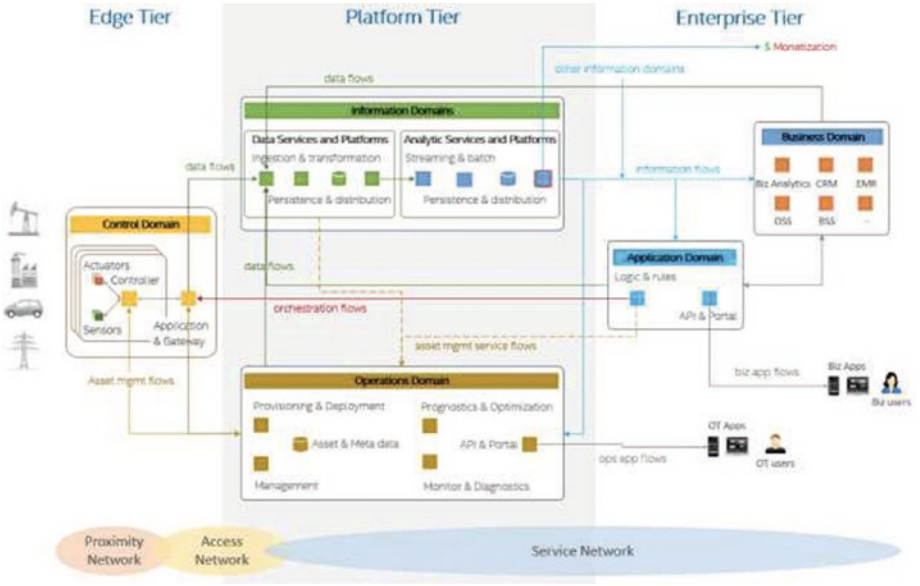

Figure 2-24. *Architectural overlay of functional and implementation viewpoints*

Open Platform Communications-Unified Architecture (OPC-UA)

Object Linking and Embedding (OLE) is a Microsoft technology aimed at office automation largely based on Windows operating systems. The Open Platform Communications (OPC) task force extended OLE for machine-to-machine control and industrial automation. The task force formed the OPC Foundation[43] in 1996 to maintain the OPC standard. OPC originally was based on Microsoft Windows-only COM/DCOM technology which was integrated with the existing OPC communications framework, resulting in a unified architecture called OPC-UA.

An industrial IoT network is really a layering of multiple networks customized to address a particular aspect of industrial operations. A typical IIoT system will consist of a four-layer system of networks (Figure 2-25). The device-level network consists of sensor-actuator devices with real-time control of physical world processes, logistics, and mechanics. The protocols linking nodes at this layer are typically traditional brownfield technologies such as ProfiNet, EtherCAT, and Modbus. These systems are designed to operate autonomously taking into consideration safety and reliability.

The control-level network consists of shop floor controllers that coordinate the end-to-end flow of the industrial system. The output of one shop floor device may be consumed as input to another shop floor device. Shop floor controllers orchestrate the hand off the work item, whether physical, informational, or both. OPC-UA is a framework for shop floor machine control. Controllers host multiple device nodes, run real-time or near real-time operating systems, and support both fieldbus and a traditional Internet protocol stack based on IP and TCP.

[43] www.opcfoundation.org/

The third level is the Manufacturing Execution System (MES) that provides plant-, site-, or factory-level coordination of various shop floor networks. This network consists of PCs and servers networked using traditional IP networks. The fourth level focuses on Enterprise Resource Planning (ERP) functions that filter data from the MES level for deeper analytics relating to process improvement, cost optimization, and operational efficiency improvement. ERP applications may be hosted in an enterprise data center or a cloud hosting environment such as Microsoft Azure.

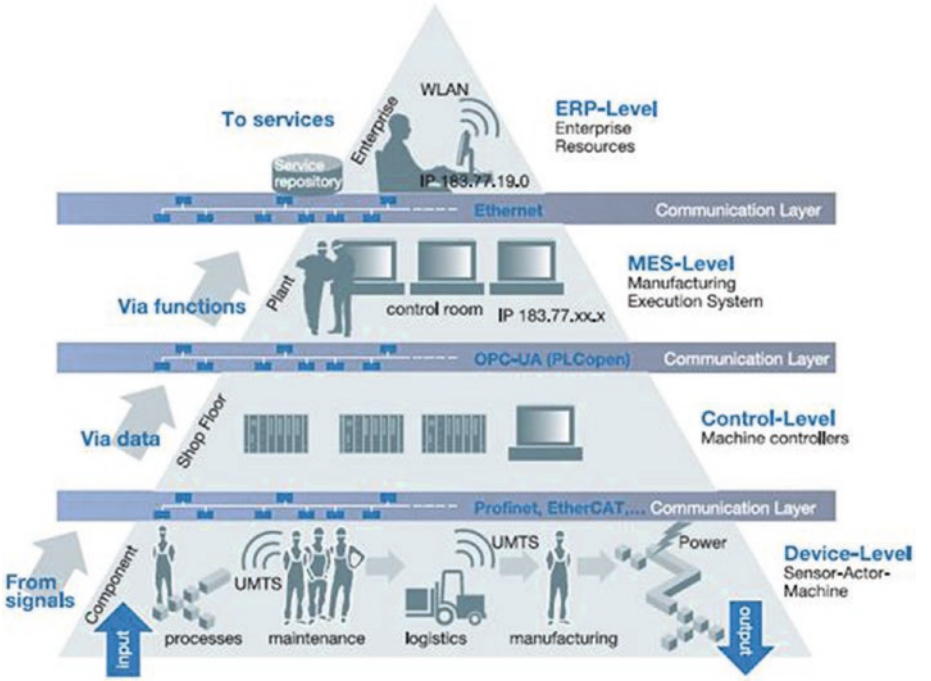

Figure 2-25. *A four-layer system of networks for IIoT with an OPC-UA layer*

OPC-UA is a device-centric technology that connects sensor, actuator, and PLC (programmable logic controller) devices to each other and to a larger system of PC and server class platforms. It aims to ensure device-level interoperability.

The basic structure of an OPC-UA network consists of an OPC-Client connected to an OPC-Server. The OPC-Server connects to sensor, actuator, and PLC devices. The OPC-Client to OPC-Server connection is typically based on IP networking. The OPC-Server to control devices is typically based on a fieldbus technology.

OPC-UA Framework Architecture

The OPC-UA design goals aim for platform independence, functional equivalence, and data interoperability through information modeling, extensibility, and security. Platform independence is achieved by porting the OPC-UA framework layer to multiple operating systems (e.g., Microsoft Windows, Apple OSX, Android, Linux) and hardware platforms based on X86, ARM, PLC, and others. As long as there is a framework instance that runs on the OS and hardware of interest, IIoT device interoperability exists.

Functional equivalence is the idea that OPC-UA applications operate consistently regardless of which operating system and hardware platform was used. There are six areas of functional equivalence defined:

(1) Discovery: Devices search for peer devices, servers, and networks the OPC-UA application needs to perform its function. Plug-and-play behavior can be supported but requires application involvement to anticipate the type of objects and operations needed.

(2) Address space layout: Devices implement a hierarchical object model where files and folders contain data that can be read/written across the network from one node to another.

(3) Access control: Data objects have access control policies that control reading and writing on a per node basis.

(4) Subscriptions: Client nodes can subscribe to data objects monitoring and receiving updates to data that changes. Client nodes may specify filtering criteria that are applied to monitored data values when determining when it is appropriate to notify the client.

(5) Events: Client nodes can receive asynchronous responses when data values satisfy a specified criterion.

(6) Methods: Client nodes execute subroutines based on server-defined criteria.

Information models define data access semantics. Each information model is independent from other information models, meaning each model has different access control, state, and quality contexts. The OPC-UA framework has several built-in information models (Figure 2-26): Data Access (DA), Alarms and Conditions (AC), Historical Access (HA), and Programmable state machines (Prog). The Data Access model supports live (near real-time) access to sensor data. Each data element has a *name* and *value*. There is also a *timestamp* to indicate when the data was read and a *quality* component that determines if the data is valid.

Historical Access (HA) data is not real-time data, and there could be a deep history of values stored. SCADA and other systems support devices that monitor sensor readings over a longer period of time. HA objects can transfer historical data from sensor to framework node easily. Framework application may apply analytics to HA data to gain additional insights into operations over a period of time.

Alarms and Conditions (AC) data doesn't have a current value. Rather it maintains subscriptions to other data where subscribers may specify conditions in which to send notifications and updates. Notifications have a *timestamp* but do not have *name* and *quality* attributes.

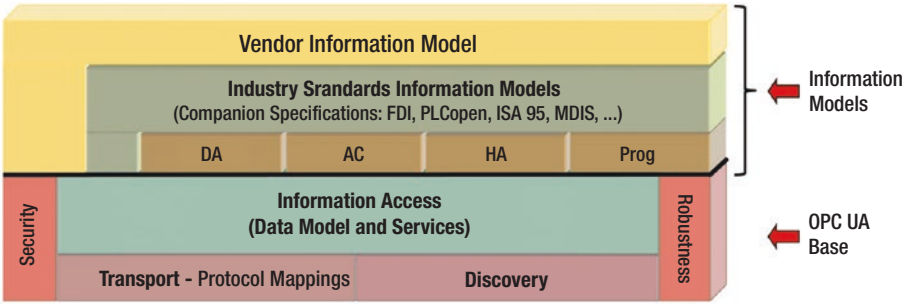

Figure 2-26. *OPC-UA information modeling framework*

Extensibility is achieved through a multilayered information model that supports vendor-specific, industry standard data models and native OPC-UA defined data models. Companion specifications define what information is exchanged, while OPC-UA Information Access layer defines how information is exchanged.

OPC-UA Security

Security is built around two framework layers (Figure 2-27). The **session layer** addresses user authorization, authentication, and access control based on role and permissions. The **secure channel layer** provides message encryption and integrity protection when exchanged between nodes. It also can be used to authenticate applications that connect with the OPC-UA framework. The security channel layer relies on TLS (Transport Layer Security) using HTTPS. Though HTTP is also supported. OPC-UA relies exclusively on X.509v3 certificates to authenticate and authorize users and applications.

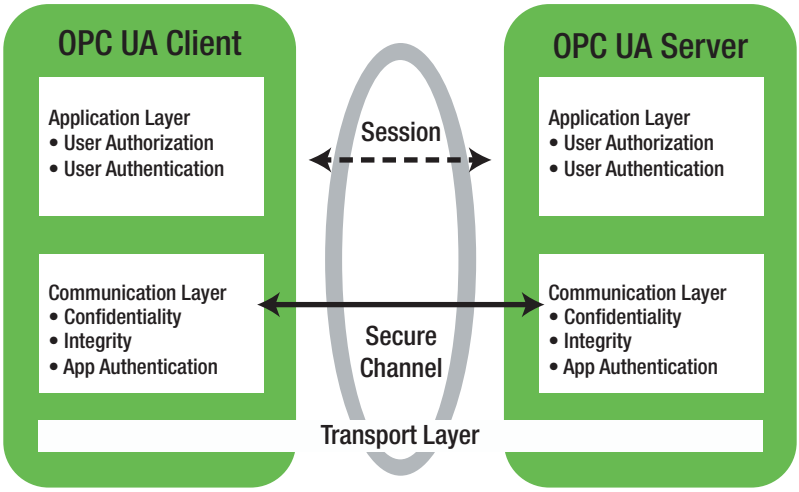

Figure 2-27. *OPC-UA secure communications*

Auditing is also supported in OPC-UA security supporting forensic investigation.

OPC-UA applications undergo a two-step access process where they first access servers and second access data. Authentication policy is expressed in terms of server or client identity, while data access is expressed in terms of read/write permissions on data objects.

The German government BSI (Bundesamt für Sicherheit in der Informationstechnik) did an extensive security evaluation of OPC-UA to determine if it is safe for using in German industry. Their conclusion was that it was designed with a focus on security and does not contain systemic security vulnerabilities. This is an important observation because, unlike other framework approaches we've reviewed, security was integral to the framework design.

However, the way in which hardware security capabilities such as secure storage, cryptographic algorithm implementation, and trusted execution environment enforcement are left as an exercise to implementers. Given the platform independence design goal, it is possible if not likely different platforms hosting OPC-UA frameworks could have very different attack

resistance properties. At the time of this writing, OPC-UA did not implement attestation mechanisms that describe implementation choices linking framework security to hardware and platform security.

Data Distribution Service (DDS)

The Data Distribution Service[44] (DDS) is a connectivity framework designed for industrial process control. It is standardized through the Object Management Group[45] (OMG) founded in 1989. The OMG is an industry standards consortium that produces and maintains specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments.

DDS v1.0 was published December 2004. DDS v1.4 was published March 2015. Companion specifications relating to security, remote procedure call (RPC), and other topics are continually updated. There are several proprietary and open source implementations of DDS. OpenDDS[46] is a popular open source implementation.

The primary design goal is summarized as the efficient and robust delivery of the right information to the right place at the right time. To accomplish this, a data-centric publish-subscribe (DCPS) approach was taken. The target applications expect the DCPS framework to be high-performance, efficient, and predictable. To accomplish these goals, DDS (a) allows middleware to preallocate resources to minimize dynamic resource allocations, (b) avoids properties that require the use of unbounded or hard-to-predict resources, and (c) minimizes the need to make copies of the data. DDS is a strongly typed system, meaning the programmer directly manipulates constructs that represent data. Interfaces are safer due to rigorous type checking, and execution code is more efficient because type checking enforcement is done at compile time.

[44] www.omg.org/spec/category/data-distribution-service/

[45] www.omg.org

[46] http://opendds.org

DDS consists of four main entities:

- **Domains** (Figure 2-28): Define a global context in which data, data readers, and data writers have ubiquitous access. The domain defines the naming scope for identifiers. Cross-domain interactions may require disambiguation using a domain identity.

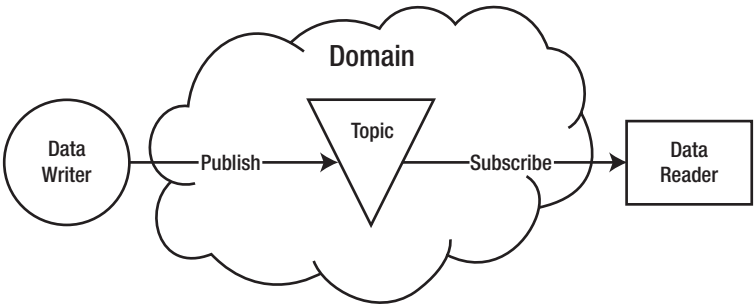

Figure 2-28. *DDS publish-subscribe data model*

- **Topics** (Figure 2-28): Are objects that conceptually fit between data writers and data readers. They define the context in which publish-subscribe interactions may take place. Topic names are unambiguous within the domain and contain a type and QoS component (Figure 2-29). Type and QoS attributes apply to the data referenced via the topic context. QoS attributes are themselves DDS Topics. Topics allow expression of both functional and nonfunctional information.

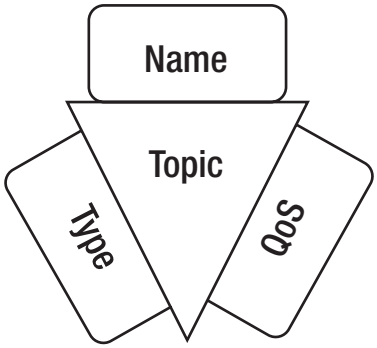

Figure 2-29. *DDS Topics have QoS integration*

- **Data Writers**: Correspond to publishers of a publish-subscribe interaction pattern and must create a Publisher instance object in order to accept subscribers or to prepare and publish data. Data writers communicate data to its publisher to initiate a publication.
- **Data Readers**: Correspond to the subscribers of a publish-subscribe interaction pattern and must create a Subscriber instance object in order to register to receive publications. Data readers communicate interest in a topic to initiate subscription registration.

Quality of Service (QoS) is a fundamental design consideration that is intimately integrated into the DDS object model. Each topic may consist of multiple data values distinguished by a key value. Different data values with the same key value represent successive values for the same data instance (e.g., a temperature sensor may maintain a short history of temperature values sensed over an interval). Different data values with different key values represent different data instances (e.g., multiple temperature sensors). QoS and type attributes apply to data instances. QoS interactions follow a *requested-offered* pattern where a data reader requests a particular QoS policy and the data writer tries to accommodate the request.

The overall flow of a DDS interaction begins with domain participants (readers and writers) joining a domain (Figure 2-30). Publishers produce data to a data partition object, while subscribers retrieve data from the data partition object. Data writers offer a QoS promise on published data based on the data reader's requested QoS level.

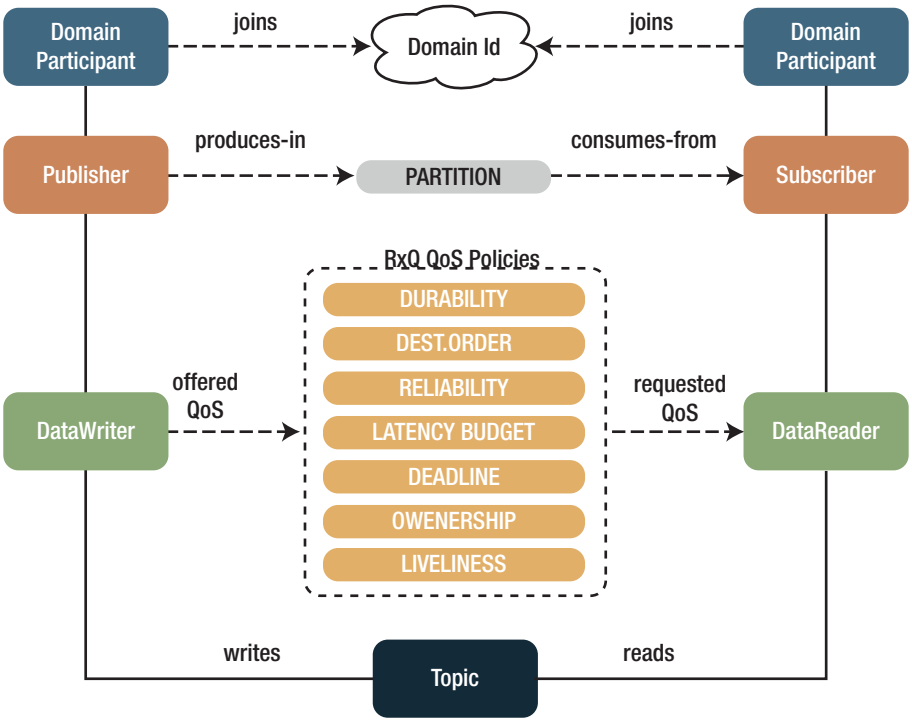

Figure 2-30. *DDS data interaction flow*

The DDS standard defines the set of possible QoS policies. These include the following QoS types:

- USER_DATA: Allows the application to attach additional information to the data object so that remote entities can obtain additional context that relates to application-specific purposes. This aids in refining

discovery queries and allows selection of appropriate security credentials or enforcement of application-specific security policies.

- TOPIC_DATA: Allows the application to attach additional information to the topic object to facilitate discovery for application-specific purposes.
- GROUP_DATA: Allows the application to attach additional information to the Publisher or Subscriber entity so that application-specific policies may regulate the way data reader listeners and data writer listeners behave.
- DURABILITY: Allows data to be read or written even when there are no current subscribers or publishers. Multiple degrees of data volatility can be defined.
- DURABILITY_SERVICE: Allows configuration of a service that implements durability attributes.
- PRESENTATION: Controls the scope of access given various data interdependencies. Coherent_access controls whether the service will preserve groupings of changes made by a publisher. Ordered_access controls whether the service will preserve the order of changes. Access_scope controls the scope of access in terms of data instance, topic, or group.
- DEADLINE: Controls the interval in which a topic is expected to be updated. Publishers must supply updates within the deadline interval, and subscribers can set a timer to check for most recent updates based on the interval.

- LATENCY_BUDGET: Allows applications to specify the urgency of the message by specifying a latency duration.
- OWNERSHIP: Controls how data writer objects interact with published data. *Shared* access means multiple writers can update the data item. *Exclusive* access means only one writer can update it. SHARED-EXCLUSIVE means multiple updaters coordinate their updates.
- LIVELINESS: Controls mechanisms for determining if network entities are still "alive."
- TIME_BASED_FILTER: Allows data readers to see at most one change to a topic at a minimum periodicity.
- PARTITION: Allows a logical partition inside a "physical" partition. Physical partitioning may have safety and security benefits, while logical partitions may have performance benefits.
- RELIABILITY: Allows reliability to be defined in terms of levels, BEST_EFFORT being the lowest and RELIABLE being the highest.
- TRANSPORT_PRIORITY: Allows alignment with transport layer QoS capabilities.
- LIFESPAN: Allows specification of when a data value becomes stale.
- DESTINATION_ORDER: Controls how each subscriber resolves the final value of the data instance when written by multiple writers.

- HISTORY: Controls when data instance changes before it is communicated to data readers. KEEP_LAST means the server keeps the most recent update. KEEP_ALL means the server will attempt to deliver all instances of changed data.
- RESOURCE_LIMITS: Controls how many resources can be applied to achieve quality of service objectives.
- ENTITY_FACTORY: Controls the flexibility of nodes in their ability to replicate or produce additional entity instances.
- DATA_LIFECYCLE: Controls how persistent or temporal data are relative to the availability of either the data writer or data reader.

DDS QoS design is one of its features that most distinguishes it from other IoT and IIoT frameworks. QoS mechanisms have both safety and security implications in that they improve data integrity – goals common to both disciplines. QoS mechanisms must be implemented in ways that ensure the integrity of the QoS system. Otherwise, the expected quality of service is suspect. Hence, trustworthy implementation of the DDS framework is essential to realizing the QoS richness anticipated by its designers.

DDS Framework Architecture

The DDS framework layering (Figure 2-31) consists of several layers beginning with an IP network layer. TCP and UDP transports make up the next layer followed by the DDS Wire Protocol for Real-Time Publish-Subscribe (DDSI-RTPS) layer. The DDS layer defines the data model abstractions described earlier. The DDS framework defines several vertically integrated technologies for security, remote procedure call (RPC), and extensions to its data typing system.

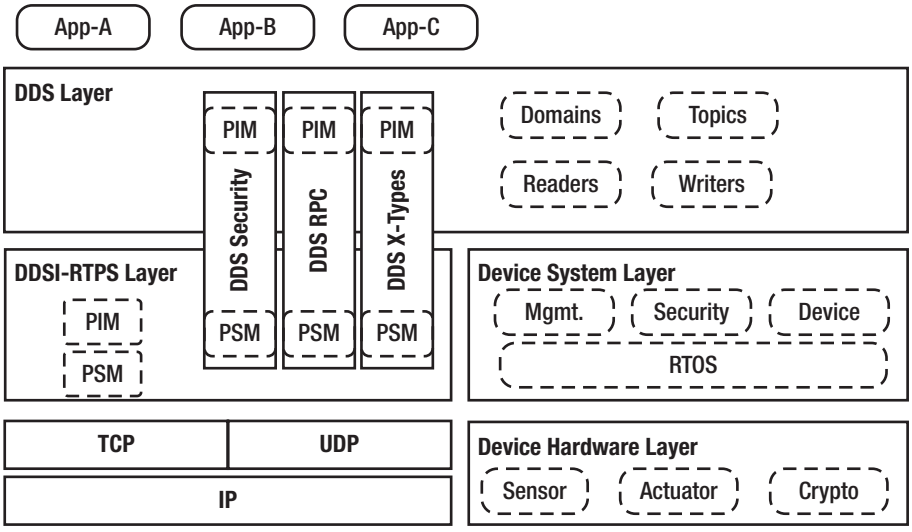

Figure 2-31. *DDS framework layering*

Implied by the DDS layering architecture is a Device System layer that implements the IoT device capabilities including native security and manageability capabilities. These capabilities depend on a Device Hardware layer that must have ties to the actual sensor, actuator, security, or other hardware features. The Device System layer exposes native device capabilities to the DDS framework through available interfaces. Different DDS framework implementations may make different implementation choices regarding how to best integrate the framework with a specific device.

The DDS specification helps isolate platform-specific elements of DDS from platform-independent elements by specifying a platform-independent model (PIM) and a platform-specific model (PSM) of DDS structures. The PSM definition ensures porting efforts result in minimal impact to the semantics and operation of the PIM while still allowing quality integration with the native platform.

The PIMs and DDS layer ensure DDS applications can expect a consistent environment for sharing information that is strongly typed and syntactically interoperable. A summary of DDS application properties is as follows:

- Applications can autonomously and asynchronously read and write data that is decoupled spatially and temporally.
- DDS data is loosely coupled due to virtualized data spaces that are designed for scalability, fault tolerance, and heterogeneity.
- As with all distributed systems, the data model must consider a data consistency model. DDS defines *data spaces* that tolerate inconsistent data but *eventually* becomes consistent. Data readers will eventually see a write but may not observe it at the same time.
- DDS discovery model isolates discovery from network topology and connectivity details so that applications may focus on data objects that are most relevant to application objectives.
- The DDS data model allows location transparency since topics, data readers, and data writers are conceptually separated from the underlying physical devices and network nodes. Integration across Cloud, enterprise, plant and mission control, shop floor, or device networks doesn't require redefinition of data syntax and semantics.
- DDS data spaces (aka domains) are decentralized. A DDS system may host multiple data spaces that involve readers and writers from any data space. There is no central point of failure.

- Connectivity among DDS entities is adaptive, meaning connections can be established and torn down dynamically. The underlying communications infrastructure can optimize for the most efficient data sharing approach.

DDS domains have global data space (Figure 2-32), meaning topics are visible to all data writers and readers that are members of the same domain. Data writers and readers may be members of multiple domains simultaneously to allow interaction with topics from different domains. It is even possible to construct a domain broker that gives the illusion of the same topic appearing in separate domains.

DDS domain interactions can become rather complex. This complexity may be especially appreciated when an access control policy is needed that places restrictions on various data writer and data reader interactions that span multiple domains.

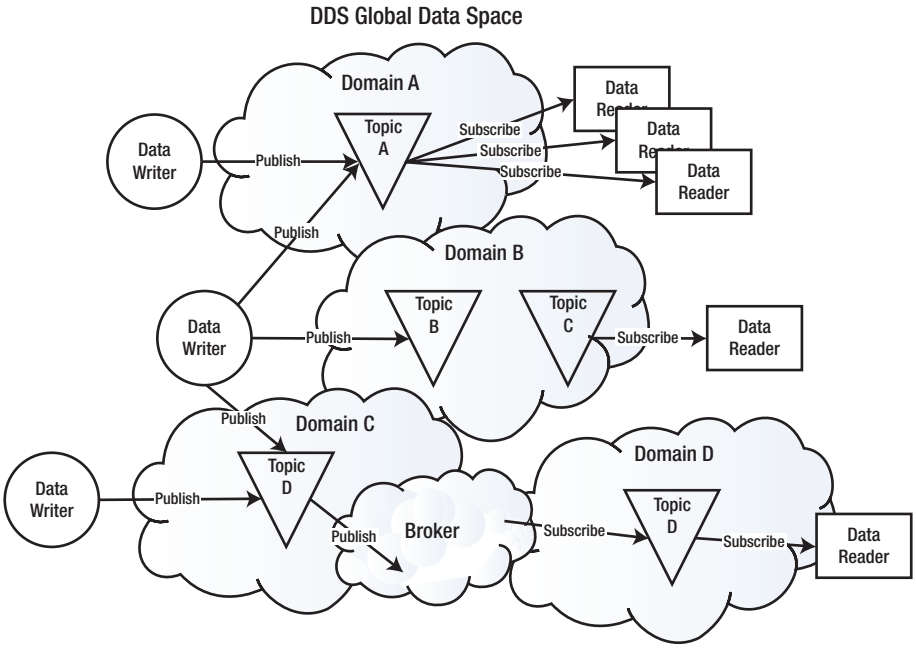

Figure 2-32. *DDS Global Data Space example*

DDS Security

DDS security consists of three main elements (Figure 2-33): (1) RTPS messages with security enveloping structures, (2) token-based security context, and (3) pluggable security modules.

Security Enveloping

Security is closely integrated into the DDS data model. Cleartext DDS data messages are encapsulated within DDS enveloping structures that support encryption, integrity, authorization, and authentication. The RTPS system uses the security enveloping structures as its main messaging structure so that the real-time publish-subscribe optimizations are preserved even when security is applied.

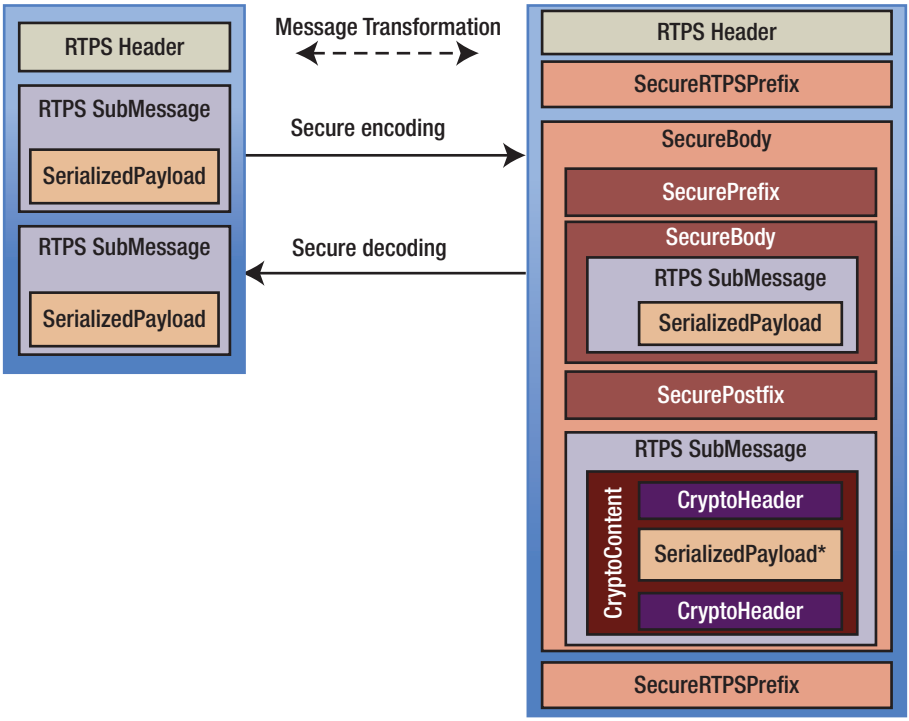

Figure 2-33. *RTPS message encoding/decoding with secure encapsulation*

A cleartext RTPS message consists of an RTPS header and one or more RTPS submessages each containing a serialized payload. To prepare a cleartext message for delivery over an unsecure channel, the cleartext message must be transformed into a secure RTPS message. Figure 2-32 illustrates the transformation. Integrity-protected RTPS submessages are wrapped by a secure body and have a secure prefix and secure postfix component. The prefix defines the integrity protection mechanism, security context, and algorithms. The secure postfix contains a hash or signature of the secure body. If the RTPS submessage requires confidentiality protection, the serialized payload of the submessage is encrypted, forming a CryptoContent element consisting of a CryptoHeader and CryptoFooter. The CryptoHeader defines the encryption method, security context, and algorithms. The CryptoFooter contains the ciphertext version of the serialized payload. All the RTPS submessages belonging to the RTPS message are bound together using another layer of security enveloping consisting of SecureRTPSPrefix, SecureRTPSPostFix, and SecureBody elements. The second layer of security enveloping ensures submessages can't be omitted, appended, or substituted by an attacker.

Security Tokens

All of the privileges obtainable by DDS entities are described using a *security token* data structure. There are tokens that facilitate secure discovery, participant permissions, and secure message exchange. Security tokens allow exchange of security information using the DDS messaging capability.

- **Discovery tokens**: Facilitate establishment of security contexts for subsequent secure interactions. The *IdentityToken* contains summary information of a domain participant in a manner that can be externalized and propagated using DDS discovery. The *IdentityStatusToken* contains authentication information of a domain participant in a manner that

can be externalized and propagated securely. The *PermissionsToken* contains summary information on the permissions for a domain participant in a manner that can be externalized and propagated over DDS discovery.

- **Permissions tokens**: The PermissionsCredentialToken encodes the permissions and access information for a domain participant in a manner that can be externalized and sent over a network. It is used by the access control plugin which manages domain access and specific reader-writer interactions.
- **Message tokens**: The *CryptoToken* contains all the information necessary to construct a set of keys to be used to encrypt and/or sign plain text transforming it into ciphertext or to reverse those operations. The *MessageToken* is a superclass of several message tokens used to maintain security context when multiple message exchanges are required such as authentication and key exchange protocols.

Security Plugin Modules

The DDS framework takes a modular approach to security so that platform-specific capabilities can be exposed to and utilized by DDS entities. There are five pluggable security modules (Figure 2-34): (1) authentication, (2) access control, (3) cryptography, (4) logging, and (5) data tagging.

- **Authentication plugin**: The principal joining a DDS domain must authenticate to a domain controller, and peer DDS participants may be required to perform mutual authentication and establish shared secrets.

- **Access control plugin**: Decides whether a principal is allowed to perform a protected operation.
- **Cryptography plugin**: Generates keys and performs key exchange, encryption, and decryption operations. Computes digests and verifies message authentication codes. Signs and verifies signatures on messages.
- **Logging plugin**: Logs all security relevant events.
- **Data tagging plugin**: Adds data tags for each data sample.

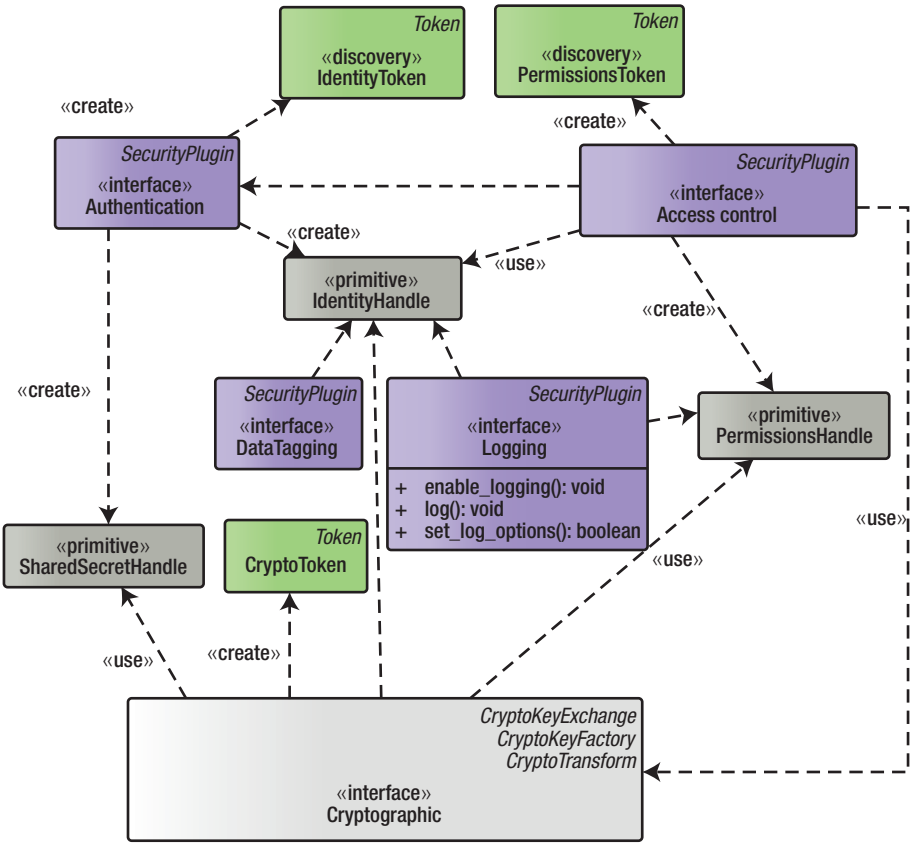

Figure 2-34. *DDS security plugin module architecture*

DDS security offers a comprehensive well-integrated security solution that aligns well with DDS design philosophy focusing on data and publisher-subscriber interactions. Security is modular, enabling platform-specific services and hardware to be effectively utilized and incorporated.

DDS quality of service parameters though originally designed to meet industrial safety requirements may also help achieve security objectives. The OWNERSHIP and PARTITION QoS parameters capture expected data sharing and partitioning semantics. Security mechanisms used for data isolation and protection may be useful toward meeting these quality expectations. LIFESPAN and HISTORY properties describe data persistence characteristics that inform regarding object reuse requirements and which data may require stronger confidentiality and integrity protection.

However, DDS goals toward heterogeneous operation make assumptions regarding the quality and condition of security plugins. An attacker might easily compromise the plugin or spoof the plugin interface allowing an attack plugin to take control. Peer nodes are not easily able to detect such attacks. For example, DDS doesn't appear to support attestation protocols that would query a peer principal's security subsystem to provide proof of device provenance and integrity of the system firmware, software, plugins, and DDS framework layers.

Framework Gateways

This chapter has focused almost exclusively on open standard IoT framework solutions, some of which have been omitted here for brevity. There are tens if not hundreds of *brownfield* frameworks with varying degrees of openness and standardization, but many are specific to an industry vertical. Cloud-connected IoT is another category of IoT framework integration mostly ignored here as well. Although many of the open standard frameworks claim interoperability with cloud

environments, the IoT cloud ecosystem largely takes a *walled-garden* approach.[47] Most have a proprietary IoT framework or support both a proprietary and open framework solutions with integration to their proprietary cloud back end. Some of these include Amazon Web Services (AWS) IoT, Apple Homekit,[48] Bosch IoT Suite, Cisco IoT Cloud Connect, General Electric Predix, Google Cloud, IBM Watson Cloud, Microsoft Azure, Oracle IoT Cloud, Salesforce IoT, Samsung ARTIK Cloud Services,[49] and SAP IoT Platform. Dell's EdgeX Foundry[50] takes a slightly different approach enabling services at the *edge*, where edge refers to both the edge of the IoT network and the edge of the cloud hosting environments. The ecosystem that traditionally supplies the *pipe* between IoT device and Cloud is interested in *moving up* the IoT stack to add more value. IoT framework technologies help enable that mobility.

The IoT framework standards organizations seem to understand that a multitude of "standard" IoT frameworks hinders one of the main motivations for IoT frameworks – interoperability! Industry efforts to consolidate frameworks have taken place already. The AllSeen Alliance and UPnP Forum have merged with the Open Connectivity Foundation. The OpenFog Consortium joined forces with the IIC and the IPSO Alliance was acquired by the Open Mobile Alliance (OMA) to form OMA SpecWorks.[51] Collaborations between framework standards organizations also help resolve interoperability challenges. For example, the OCF is thought to be working on an OCF[52] to OneM2M bridge[53] (aka framework gateway).

[47] www.electronicdesign.com/embedded-revolution/iot-frameworks-ties-bind

[48] https://developer.apple.com/homekit/

[49] https://artik.cloud

[50] www.edgexfoundry.org

[51] www.businesswire.com/news/home/20180327005208/en/IPSO-Alliance-Merges-Open-Mobile-Alliance-Form

[52] https://wiki.iotivity.org/bridging_project

[53] https://openconnectivity.org/draftspecs/Cleveland/CR2595_Cleveland_Bridging_Security_20181004.pdf

But these efforts are solutions to an interoperability problem created by the industry's eager response to an IoT interoperability problem. Ironically, the "success" of IoT seems to have created a more complex environment for IoT interoperability as both standard and proprietary "connectivity" frameworks and toolkits proliferate. Framework gateways naturally come to the rescue, but at what cost to usability, manageability, and security?

Framework Gateway Architecture

This section outlines several approaches for gatewaying (aka bridging) IoT frameworks, considers security implications of each, and suggests an idealized architecture for secure IoT framework gateways.

Type I Framework Gateway

A type I framework gateway (Figure 2-35) combines unmodified framework gateways using a common framework gateway application. The application performs all necessary object model translations and data structure mappings to achieve interoperability. The application (i.e., developer) must have intimate understanding of data object syntax and semantics for both (all?) sides of the translation. Some objects in a first IoT network may not have a suitable corresponding counterpart (sensor, actuator, controller) in the other IoT network for the applications to simply "wire" them together. Instead, it must create an abstraction that approximates an object that is recognizable and considered to be a safe alternative interaction. For example, a dimmable light bulb in Network A may support 10 levels of brightness, while a dimmer control in Network B supports 100 levels of control. The gateway application provides the mapping function that divides by 10 in one direction and multiplies by 10 in the other direction. In some cases, there may not be a reasonable mapping, and the gateway application developer may take some other

approach such as exposing the devices to a console interface so that a user can resolve any mapping conflicts or ambiguities. Polyglot[54] is an example technology that aids in the development of type I IoT framework gateway applications.

Type II Framework Gateway

A type II framework gateway (Figure 2-35) expects the network Connectivity, Node Interaction, and Data Object layers are dissimilar, but there is a Data Object layer mapping object that relates Framework A objects with Framework B objects. A gateway application supplies administrative control such as installing, updating, and monitoring an object translation component that exists within the Data Object layer. Typically, designers of each interoperating framework must collaborate to identify semantically similar but syntactically dissimilar elements and their mapping functions. The design collaboration may reveal disconnected design semantics as well that may be clarified in related gateway-specific specifications or may result in specification revisions that clarify ambiguities. For example, one framework might expect all objects to be discoverable through its hosting endpoint device, while another framework might expect discovery is handled using a dedicated discovery service. The object translation layer defines the framework-specific discovery conventions so that endpoint devices can function unmodified. This might involve having the gateway device advertising itself as a discovery service operating on behalf of devices represented in a foreign network. The OCF-AllJoyn bridging specification[55] is an example type II framework gateway that supports bidirectional bridging and device

[54] https://github.com/UniversalDevicesInc/polyglot-v2

[55] https://openconnectivity.org/specs/OCF_Bridging_Specification_v1.3.0.pdf

interactions within a common operational domain. See the“Security Considerations for Framework Gateways” section for more insight on interdomain bridging considerations.

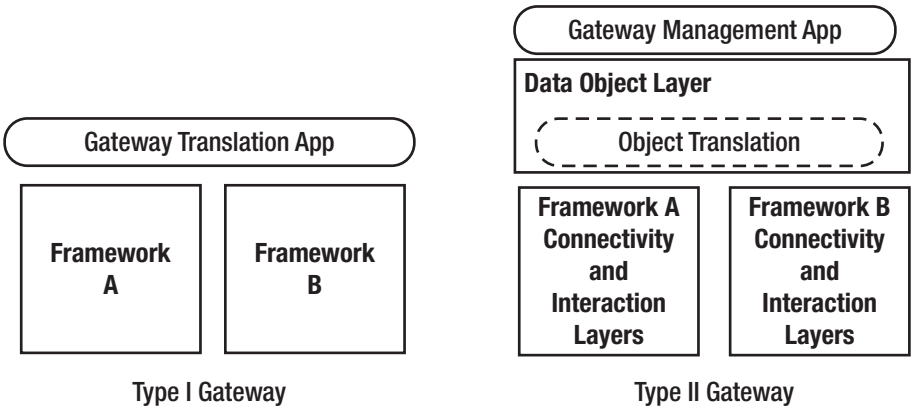

Figure 2-35. *Layering architecture for type I and type II framework gateways*

Type III Framework Gateway

A type III framework gateway (Figure 2-36) anticipates a common data object layer is in place. However, because the lower layers are dissimilar, not all data objects will be common. Therefore a data object translation capability is also required. The Connectivity and Node Interaction layers are dissimilar, but there is a message translation model that relates the interface definition model for Framework A to the interface definition model for Framework B. An example message translation operation might relate publish-subscribe messages defined by Message Queuing Telemetry Transport (MQTT)[56] to the publish-subscribe model defined by eXtensible Messaging and Presence

[56]MQTT Version 3.1.1. Edited by Andrew Banks and Rahul Gupta. 29 September 2014. OASIS Standard. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

Protocol (XMPP).[57] Another example mapping technology is the Internet Engineering Task Force (IETF) OSCORE[58] specification that maps HTTP message security to CoAP messages and vice versa.

A traditional framework may not be regarded as a type III gateway depending on the set of protocols and message types the framework supports. If a framework includes support for both HTTP and CoAP, for example, then mapping between may be a normal IoT framework function. However, given Framework A support for only HTTP and Framework B support for only CoAP, the type III gateway translation comes into play.

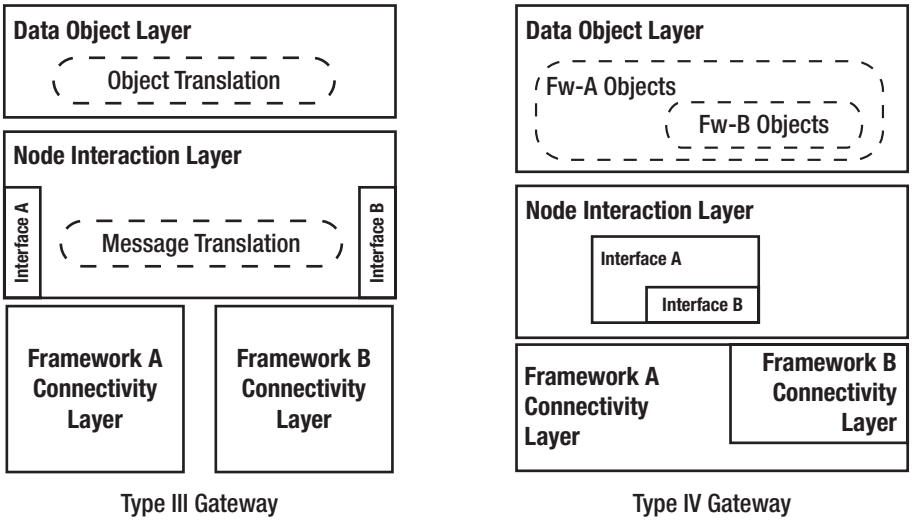

Figure 2-36. *Layering architecture for type III and type IV framework gateways*

[57]Internet Engineering Task Force (IETF) RFC 6120, March 2011. https://xmpp.org/rfcs/rfc6120.html

[58]Internet Engineering Task Force (IETF) "draft-ietf-core-object-security-15," Expires March 4, 2019. https://datatracker.ietf.org/doc/draft-ietf-core-object-security/

Type IV Framework Gateway

The fourth framework gateway class, type IV, considers the case where Framework A is a superset of Framework B. The superset and subset frameworks remain unmodified, but applications may interact with devices from either framework seamlessly. The gateway function exists when Framework A objects are exposed to Framework B and when Framework A peers are different from Framework B peers. Though subtle, this is a system boundary crossing that requires security controls. An example of this scenario is OneM2M where LWM2M supplies the device management capabilities for a OneM2M framework. Nevertheless, LWM2M also may stand alone as an independent IoT framework. The type IV framework gateway has an object model where the Framework A object model is flexible enough to encompass the Framework B object model. Likewise, the interface definitions in the Node Interaction layer have a superset-subset relationship, and the connectivity layers are similarly encompassing. The gateway function may be provided as an application of the framework or may have embedded mapping operations. The OCF framework resource naming specification allows resources to be identified using a Uniform Resource Identifier (URI)[59] of arbitrary nesting depth. A LWM2M object identifier is a URI that is constrained to two layers of nesting, and object names are numeric. The LWM2M namespace fits within the OCF namespace; hence an OCF to LWM2M gateway function could be implemented.

[59]Internet Engineering Task Force (IETF), RFC 3986, January 2005. https://tools.ietf.org/html/rfc3986

Security Considerations for Framework Gateways

Framework gateways may facilitate interdomain interactions in addition to facilitating interoperability between dissimilar IoT frameworks. Security at the framework gateway should address at least two important security questions: (1) Does the gateway bridge network domains and to what extent is the gateway trusted to perform these duties? (2) Where in the framework layering do authentication, authorization, integrity, and confidentiality protections begin and end for a given message transiting the gateway?

The Industrial IoT Consortium (IIC) describes brownfield-greenfield security integration in terms of security gateways (Figure 2-37). In this model, the gateway occupies both an interoperability and a security function. Legacy IoT endpoints may enjoy intra-brownfield interactions (often without native security), but when protocol directs interaction with the Secure Endpoints, the Security Gateway must augment legacy messages with message protections. This entails encrypting or signing messages before the Secure Gateway forwards Legacy Endpoint messages to Secure Endpoints. It may also require authenticating Secure Endpoints before allowing them to access Legacy Endpoints.

The Security Gateway function ensures crossing a network domain doesn't weaken security. Security gateways may be expected to perform the following security operations:

- Authenticate endpoints to the gateway and gateway to the endpoints.
- Authenticate endpoints from a foreign domain to endpoints in the local domain. This may require creation of a virtual endpoint on the gateway device if interior endpoints can't support the needed security capabilities.

- Integrity and confidentiality protect messages passing through the gateway. The gateway may need to decrypt then re-encrypt using native domain's recognized security associations, security algorithms, and protocols. On rare occasion domains have all these security elements in common.
- Authorize access to objects in a local domain by endpoints from a peer domain.
- Inspect and log activity between the domains.
- Establish endpoint credentials in the peer network environment. Different domains may have dissimilar security services for authentication, authorization, and key management. The gateway may be required to host security services on behalf of a local domain so that a peer domain can utilize its chosen set of security services.
- Perform data structure translation and protocol mapping functions previously described. Modification to data objects and protocol message that are integrity and confidentiality protected necessarily implies the gateway is authorized and trusted to perform these transformations.

In general, the gateway is expected to be one of the most trusted nodes in the network. Since it connects multiple domains, it likely needs to be the most trustworthy node across all the connected domains.

To achieve the preceding security goals, a Security Object layer (Figure 2-40) is needed in addition to the framework's Data Object layer. The Security Object layer must be common to all domains that connect through the framework gateway; otherwise, there is little confidence that security for the domains is correct.

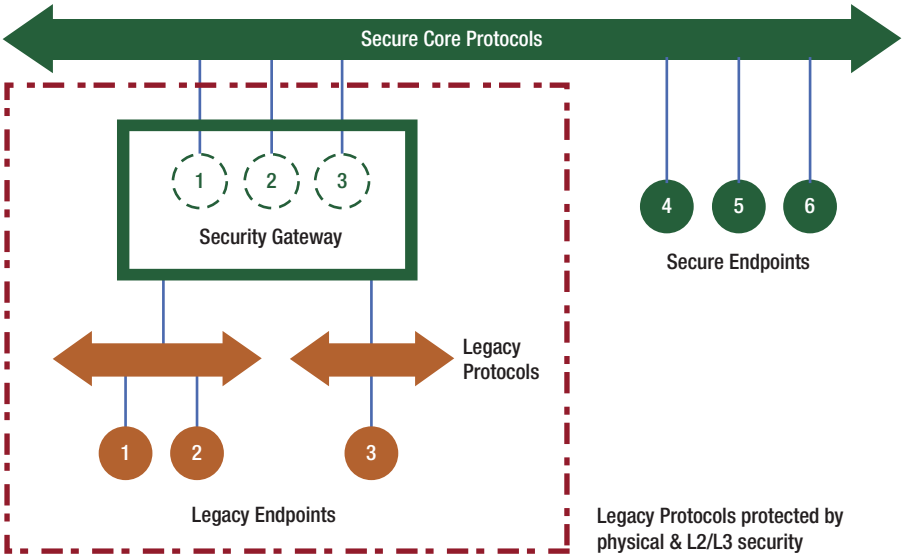

Figure 2-37. *Framework gateway as a secure endpoint/proxy to unsecure legacy endpoints*

Security Endpoints Within the Gateway

When a message enters a framework gateway, it arrives with security protections specific to its native network. Those protections terminate somewhere within the framework gateway where it is assumed the gateway will preserve the security properties throughout until the message emerges on another network where the destination network's native protections are applied. The framework gateway must satisfy authentication, authorization, integrity, and confidentiality protections in a manner that is consistent with both source and destination networks as the message transits through the gateway. The place where the network's native protection mechanism ends or begins is referred to as a *security endpoint*. The place where confidentiality protection (i.e., encryption) ends (or begins) is the confidentiality endpoint. The place where network native authorization protection ends is the *authorization endpoint* and

so forth. The various framework gateway types have different semantics and make different assumptions about security endpoint termination and origination. This section highlights some of these differences.

Security Endpoints in Type I Gateways

The security endpoints in a type I gateway (denoted by up arrow and down arrow in Figure 2-38) could in theory terminate at or near the application interface since the gateway translation and mapping functions are applied at the application level. Given a scenario where security protections are applied directly to framework objects rather than to protocols or interfaces, the data confidentiality and integrity protections may persist until the last possible moment before the framework hands off the data to the application.

Most IoT frameworks require security endpoint termination within the framework layers or in protocol layers beneath so that the framework data objects can be manipulated. This implies the data will be unprotected through some portion of framework layering before handing off to the Gateway Translation Application and again in the reverse flow. The security expectation for type I gateways is the framework architecture must strictly isolate resources belonging to Framework A from resources belonging to Framework B. Attacks originating from Framework A should be ineffective at compromising Framework B resources without first compromising the gateway or the Framework Translation Application. This simplifying assumption can be quite powerful because there are few if any exceptional cases. Exceptional cases have a tendency to expose security weaknesses that lead to exploits.

Note that within each framework context, native network operations may require *authentication endpoints* for network packet delivery that terminate within the framework. This differs from security endpoints associated with application layer message confidentiality and integrity protection.

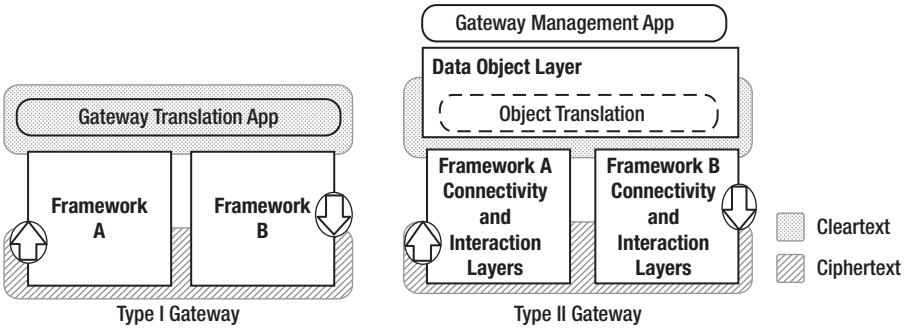

Figure 2-38. *Security considerations of type I and type II gateways*

Security Endpoints in Type II Gateways

A type II gateway requires translation at the Data Object layer implying security endpoints must exist at the base of the Data Object layer or below. The gateway application largely doesn't participate except to provide administrative oversight; hence there isn't an expectation the Gateway App should be privy to object data.

Framework A resources at the Interaction and Connectivity layers are strictly isolated from Framework B. However, because the object translation logic is shared across Network A and Network B, the Data Object layer, compromise of this layer implies access to both A and B networks. The authors feel the Data Object layer should be a third isolation environment where access to Framework A or Framework B isolation environment doesn't imply, automatically, access to the Data Object layer isolation environment. Rather, the respective isolation environments should have well-understood interfaces and semantics for crossing environment boundaries. Object translation steps necessarily invoke environment boundary-crossing primitives.

Note that in cases where framework design choices result in a security endpoint terminating in the connectivity or interaction layer, for example, if Transport Layer Security (TLS) is used for confidentiality. The isolation environment must preserve confidentiality of data as it passes between the various isolation environment boundaries.

Security Endpoints in Type III Gateways

A type III gateway (Figure 2-39) requires message protocol translation at the Node Interaction layer and may require object translation at the Data Object layer. Managing security endpoints that terminate at different layers can be tricky. If confidentiality endpoint occurs within the Data Object layer, then message translation can proceed in the Node Interaction layer since message payloads are opaque at this layer. Nevertheless, an authentication or authorization endpoint is required at this layer that authorizes a boundary crossing, for example, from Framework A to Framework B.

However, if A and B disagree on data object format, then the payload transits to the Data Object layer for object translation before it is repackaged into a Framework B message body. The Data Object layer must correctly apply confidentiality endpoint processing, possibly resulting in application of a Framework B–specific confidentiality endpoint before transitioning back to the Node Interaction layer. All of this security context must be preserved and must resist confused deputy attacks.

Isolation of respective connectivity layer environments from Node Interaction and Data Object environments seems reasonable from a security isolation perspective but appears concerning from a performance optimization perspective.

Security Endpoints in Type IV Gateways

A type IV gateway (Figure 2-39) expects data objects, interfaces, message formats, and network connectivity are a subset of the first framework. Therefore, data object, interface, and message translation might not even be needed. If it is needed, it occurs on the context of the superset framework, meaning the security endpoints that are valid for the subset framework are also valid for the superset framework. This is a nice simplifying assumption that allows for flexible isolation strategies. The point where the security endpoint begins can largely be configurable.

One important consideration is whether or not interaction with Framework B allows access of superset data objects not normally part of subset objects by Framework B. Given this scenario, the boundary crossing occurs at the line where superset and subset objects intersect. Gateway isolation mechanisms should allow separation of resources along these lines. Success or failure at applying the isolation mechanism falls largely along two vectors: (a) the degree of modularity found in the implementation of the frameworks and (b) the level of granularity with which the isolation mechanism is able to conscribe resources.

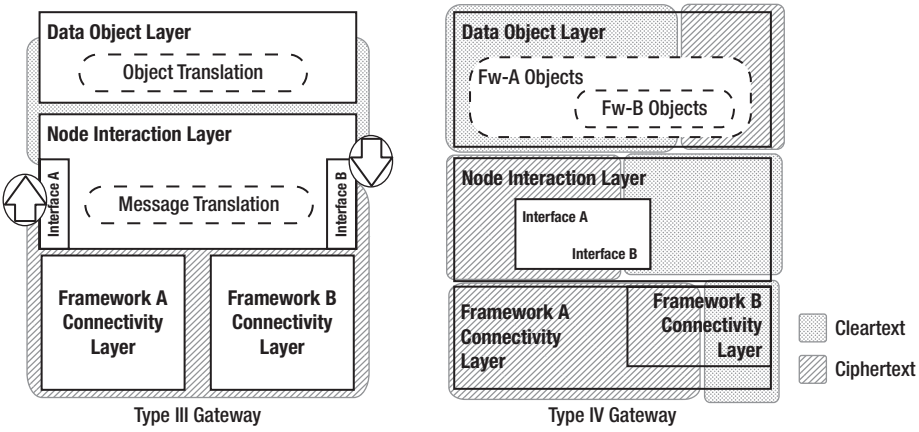

Figure 2-39. *Security considerations of type III and type IV gateways*

Security Framework Gateway Architecture

This section describes an idealized security framework gateway architecture (Figure 2-40) that more easily would support the security, isolation, performance, and flexibility requirements needed to facilitate framework gateway challenges. The meaning of an *idealized* architecture is it attempts to describe IoT framework architecture where security is central to the design and integrated from the start. It may serve as a guidepost from which to better evaluate security hardware and software solutions presented in subsequent chapters.

A prominent feature in our idealized framework architecture is the addition of the Security Object layer containing commonly understood and specified security objects and data model representations. In our experience, many IoT framework architectures cite industry standards such as X.509, TLS, and COSE in response to questions of security interoperability. However, they do not capture the semantics of what it means to be secure. There have been attempts at defining security policy languages such as XACML and SAML, but these, or something similar, have not yet been integrated into IoT frameworks.

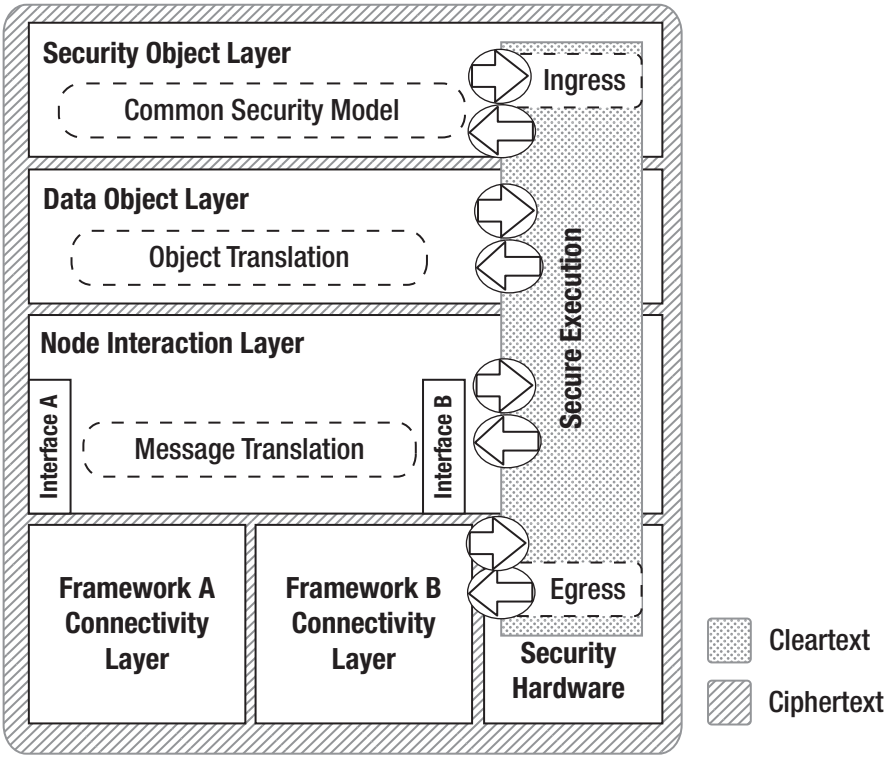

Figure 2-40. *Idealized security framework gateway*

Secure execution is another component to our idealized architecture. Secure execution is a hardware-supported mode of execution enterable when a security endpoint in the framework is required to perform security-related functions and exits upon completion. Since a security endpoint could exist at any framework layer, secure execution can be entered at any framework layer. Framework data are in cleartext while in the secure environment and ideally, confidentiality and integrity protected while outside the environment.

Framework context is maintained across ingress and egress transitions so that layer crossings can be recognized as these may correspond to network boundary crossings in a gatewaying usage context. The Security Object layer use of the Secure Execution resource preserves its isolation properties with respect to the other layers. Data passing between framework layers, which have layer isolation requirements, relies on the Secure Execution environment technology to enforce isolation requirements, these include decryption upon ingress, tenant-specific resource isolation while in the SE environment and encryption upon egress.

Although the authors are not aware of a secure execution technology that fully implements the idealized framework architecture, there are a few technologies that come close. For example, Intel Software Guard Extensions (SGX), ARM TrustZone, and virtualization have compelling potential. Chapter 3 explains in greater detail various Intel hardware security features and how they apply to IoT.

Summary

IoT frameworks occupy an important position in IoT system design as an effective strategy for empowering IoT application developers to more easily construct rich distributed IoT applications. Many of the connectivity challenges resulting from fragmented brownfield systems are hidden behind IoT frameworks. IoT applications simply expect the dissimilarities

in machine control networks, process control systems, manufacturing execution systems, and cloud integration are conveniently "simplified" for all intents and purposes.

Nevertheless, the IoT ecosystem hasn't settled on a single IoT framework technology that satisfies every industry and meets every need. Neither is there consensus over standardization of open IoT frameworks as there are multiple framework standards efforts. New and existing proprietary approaches also seem to have gained ground as the size of IoT grows. The recent proliferation of IoT frameworks, toolkits, and middleware combined with existing brownfield IoT suggests greater challenges to come for interoperable applications in a heterogeneous distributed world of IoT.

IoT framework standards organizations seem to recognize these challenges and have responded by merging organizations and standards. They have developed gatewaying and bridging technologies that let framework application interoperate through dissimilar frameworks. Noted mergers include OCF, AllJoyn, UPnP, IPSO, OMA, IIC, and OCF. There is continued interest in framework gateway interoperability among remaining frameworks, but it isn't clear that the industry needs to converge to a single or even a small number of frameworks as security, safety, reliability, and other factors may in fact motivate keeping some parts of IoT systems separated.

Framework gateways are positioned on the edges of IoT networks addressing interoperability needs but also should be considered the most trusted security control points since crossing organizational domains often coincides with translating from one IoT network protocol to another.

This section highlighted several IoT frameworks showing how various IoT system integration and interoperation requirements may be addressed. We considered challenges facing framework application interoperation in an environment of multiple frameworks. The industry's eager embrace of IoT frameworks has led to the need for framework

gateways that reassert the desire for interoperability, but also for security. We further consider ways to secure framework gateways looking at various approaches and trade-offs.

In summary, frameworks appear to offer significant value for enabling interoperable IoT applications by hiding much of the complexity of multiple connectivity technologies, messaging solutions that incorporate multiple hundreds or thousands of nodes, and data schemas that present consistent, declarative, and vendor-neutral expressions of IoT objects. We've shown that frameworks are great tools to manage IoT device complexity, but the security robustness or hardening can only be achieved by leveraging the underlying HW security capabilities dealt with in detail in Chapter 3 and are exposed via API and different framework and protocol layers by the SW as detailed in Chapter 4. The external interactions that an IoT device experiences during the lifecycle depend upon the stimulus from myriad connectivity interfaces, and this is dealt with in detail in Chapter 5.