THE RISE

of the

ROBOT OVERLORDS

CLARIFYING THE INDUSTRIAL IOT



STAN SCHNEIDER, PHD
CEO OF REAL-TIME INNOVATIONS, INC.
SUNNYVALE, CALIFORNIA

© Copyright, Real-Time Innovations, Inc. 2018. Cover figure copyright: koya79 / 123RF Stock Photo

Abstract

This eBook examines the key architectures and applications of the future of computing, aka the Industrial IoT. It examines the economic implications of the stunning rise of fast computing and networking on smart machines. It includes a practical selection guide for the most popular connectivity technologies, differentiating between DDS, OPC UA, MQTT, RESTful HTTP, OneM2M and CoAP. It then more deeply analyzes the theoretical basis for data-centric and device-centric system integration, scalable cloud-to-edge system design and security.

It concludes with a deeper look at the connectivity technology required for today's most imminent smart machines: autonomous vehicles.

About the Author



Stan Schneider is the CEO of Real-Time Innovations, Inc. (RTI). With over 1000 designs, RTI is the largest of the connectivity framework vendors in the Industrial IoT. RTI software runs the largest power plants in North America, connects perception to control in vehicles, coordinates combat management on US Navy ships, drives a new generation of medical robotics, controls hyperloop and flying cars, and provides 24x7 medical intelligence to hospital patients and emergency victims. These applications form the core of experience for this eBook.

Stan also serves as Vice Chair of the Industrial Internet Consortium (IIC) Steering Committee. The IIC is the largest of the Industrial IoT consortia with more than 250 members. RTI is also very active in the IIC serving as co-chair of the Connectivity, Ecosystem, Automotive and Data Management task groups, and leading testbeds in Connected Healthcare, Microgrids, Traffic Management and Security Evaluation. Until his election as Vice Chair, Stan chaired the IIC testbed subcommittee.

Additionally, Stan serves on the advisory board for the Industry's largest event, the IoT Solutions World Congress. He received Embedded Computing Design's Top Embedded Innovator Award in 2015 and was ranked #8 in IoTOne's Top 100 IIoT influencers in 2017. Stan completed a PhD in EE/CS at Stanford with a thesis on autonomous robotics at the Stanford Aerospace Robotics Laboratory.

CONTENTS

THE RISE OF THE ROBOT OVERLOADS: CLARIFYING THE INDUSTRIAL IOT

INTRODUCTION: ECONOMIC IMPACT 01

- THE ROBOT OVERLORDS ARE COMING 01
- BENEFITS OF SMART MACHINES 02
- IMPACT ON EMPLOYMENT 05

SOFTWARE IS EATING THE WORLD 05

WHAT IS THE INDUSTRIAL IOT, ANYWAY? 07

- CLARIFYING THE TERMS OF THE IOT 07
- THREE GREAT SPHERES OF THE IOT 07
- MARKET EVOLUTION 09

A PRACTICAL GUIDE TO IIOT CONNECTIVITY 10

- CONNECTIVITY TECHNOLOGIES DON'T OVERLAP 11
- THE INDUSTRIAL CONNECTIVITY STACK 13
- COMBINING STANDARDS: CORE CONNECTIVITY STANDARD ARCHITECTURE 15
- CHOOSING A CONNECTIVITY STANDARD 16
- COMPARISONS 27
- THE FUTURE CONNECTED IIOT 29

THEORY: SYSTEM, SOFTWARE AND INTEGRATION 30

- OPC UA 31
- DDS 33
- COMBINING OPC UA AND DDS 36

WHY A DATABUS IS SO UNIQUE 37

- THE CHANGE IN MINDSET TO DATA CENTRICITY 37
- HOW DOES A DATABUS WORK? 38
- THE DATABASE COMPARISON 41
- THE LAYERED DATABUS PATTERN 47

THE IOT SECURITY CHALLENGE 52

- SECURITY AND DATA CENTRICITY 53
- HOW DDS HANDLES SECURITY 55

A DEEPER EXAMPLE: AUTONOMOUS VEHICLES (CARBOTS) 56

- WHY DO WE NEED CARBOTS? 56
- THE CHALLENGE OF AUTONOMY 57
- AUTONOMY DEMANDS 58
- AUTONOMY ARCHITECTURE 59
- CARBOTS AND CONNECTIVITY 68

THE AGE OF ROBOT OVERLORDS 69

INTRODUCTION: ECONOMIC IMPACT



The Robot Overlords are Coming

Recently, my 14-year old son brought home a not-so-great report card. I wanted to motivate him to do better, so I asked, "What do you want to do with your life?"

He looked up at me. Then, he looked down at our dog, sleeping contentedly on the floor. He looked back at me, and said, "Dad, I want to be a pet of the robot overlords."

I was floored. Say what? This is my legacy? This is his ambition?

Then I realized...throughout history, those who leverage the technology of the day do best. He plans to specifically leverage smart machines? That's not a bad career ambition at all. Being a pet is smart. Being a good pet is brilliant.

Let's do some math. Moore's law says that computers double in capability every 1.5 years. This rate shows little sign of flattening. Assuming it continues, computers in 40 years will not be 10x better than today. Or 100x. Or 1000x. Computers will be a mind-boggling 100 million times better than today. Even if this number is off by orders of magnitude, it's shocking. The implications could not be more profound.

Your children will not drive. Your children will not write. Your children will not compose music. Your children will not make buying decisions. Or, at least they will not have to. There are already intelligent computing systems challenging human capabilities in all these areas.

The rise in raw processing power is only half of the story. Today's other mega trend is pervasive device networking, aka the "Internet of Things" (IoT). There are already several times more devices online than there are people. Soon, there will be hundreds of billions of devices connected. Easy, fast networking is following an exponential curve as dramatic as processing.

And, we are only just approaching the most dramatic acceleration of both these trends. For at least the next two or three decades, the most important technology drivers will be the rise of artificial intelligence and pervasive device networking. Soon, essentially everything man-made will be both smart and connected.

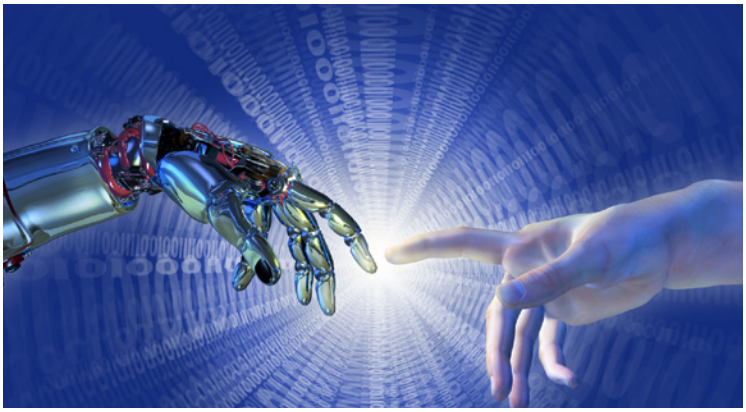


Figure 1. Disruption Gartner said the smart machine era will be the most disruptive in history. The exponential increase in processing power and networking will profoundly impact every industry.

But, industry moves slowly. For decades, little of importance has changed. In fact, most of today's industry is unchanged by the Internet. Medical treatment systems, the power grid, manufacturing lines, process control and transportation systems work today the same way they did 20 years ago. But in the next few years, clever engineers in every industry will find a way to leverage this amazing change in computing. That feat will threaten everyone and everything that does not respond. The success of every player in every industry will depend on its ability to compete in this new reality.

The rise of incredibly-powerful connected computing is not just another of N factors in design. It is the factor. This change will rewrite vendor relationships, redefine profitability, and re-imagine delivery from environment to cost to product. Today's system designs must last decades. If you are a designer, your challenge is to look beyond today's experience into a future, the future, dominated by intelligent computing.

Benefits of Smart Machines

So, we will all be pets. That's good, because the robot overlords will make the world much better.

Consider medical care in hospitals. Shockingly, hospital error is the 3rd leading cause of death in the U.S. Today's devices don't talk to each other. They don't even know they are connected to the same patient.

This system is not good enough. Without the ability to check readings between devices, it's hard to differentiate a nuisance alarm from a life-threatening condition. Simple mistakes happen in hundreds of ways. For instance, every

heart-bypass surgical team that Harvard Medical School surveyed made the same disastrous mistake at least once: forgetting to resume ventilation after disconnecting the heart-lung machine. Patient-controlled analgesia (PCA), the system that allows thousands of patients to self-administer morphine after a surgery, is widely used and considered safe. But, it still kills 2-3 patients a day. There are just far too many failure modes for harried hospital staff to do this on their own.

The smart machine disruption will change all that. By connecting devices together and adding "doctor in a box" smart software, most all these errors will be history. And, it's already happening! GE Healthcare, for instance, is building a system capable of handling 200,000 connected devices in a 1200 bed hospital. It will be deployed soon.

The environment will also benefit. Our current power grid handles only one-way power flow. Big generators produce power that flows to distributed loads in neighborhoods and companies. There hasn't been a major disruption since the advent of 3-phase transmission in the 1930's. But it's clear: the future power grid must incorporate distributed renewables like solar and wind.

How? It's not that simple. Big spinning-metal generators cannot change their output faster than about 15 minutes. So, centralized controllers tell them to generate enough power to cover the expected load over the next 15-minute period. They have to add a small margin of "spinning reserves," because if the generation ever drops below the load, the grid can fail completely.

Solar adds generation capacity, but clouds can cover entire neighborhoods in minutes or even seconds. The big spinning generators can't react that fast, so the central controller must increase spinning reserves. Unfortunately, spinning those reserves burns nearly as much oil or coal as the no-solar original system. So, a shocking percentage of solar energy is simply wasted.



Figure 2. Connected Medical Devices GE Healthcare, the largest manufacturer of hospital devices, is building the next generation of intelligent, connected devices. The diagram shows a test with 300 rooms with a hierarchical connection topology. The design will allow over 1000 rooms and hundreds of thousands of devices.

The solution? Just like the medical case, connect the devices and add intelligent software. In this case, the software wraps local control loops around the solar cells and a little local battery storage. Even 15 minutes of storage smooths the fast bumps and allows the generators to reduce reserves.

This also is already happening. Duke Energy, the largest U.S. utility, leads an effort called "OpenFMB" designed to transform the grid. Dozens of companies are cooperating to build an implementation. Intelligent, distributed control will soon sweep the industry, bringing much greener energy and much better efficiency.



Figure 3. Carbot Autonomous cars will forever change transportation. Carbots will increase free time, provide mobility for all, and eliminate most collisions.

These are great, but medicine and power are not the greatest disruptions. That distinction goes to "driverless cars." Autonomous cars are more like robots on wheels than an evolution of cars. For these reasons, I will refer to them as "carbots." There is a drastically new player in our lives. A new player deserves a new name.

The benefits are many and profound. By eliminating the cause of 94% of accidents, Carbots will make our roads significantly safer. Time wasted today can be used to sleep or work. Carbots can drive in close, efficient formation, saving energy and time and making good use of today's carpool lanes. Because we don't need to store carbots near you, we can turn urban parking lots into parks. They will make extensive travel practical for millions. They will clear DUIs from our courts. They will bestow freedom of motion on those who can't drive. They will disrupt nearly every aspect of daily life, all of it for the better.

Of course, carbots will also disrupt the 30% of U.S. jobs directly related to driving. These include the obvious: truckers, delivery people and Uber drivers. The greater effects will eliminate roadside hotels, revolutionize elder and child care, obsolete auto insurance, and do away with auto-body shops. Driverless cars will impact our economy and society more than anything since the horseless carriage.

In the end, smart machines will make our world better... much better. Your children may not drive, write, compose music or make buying decisions. But, they will live in a cleaner, healthier, safer, world. And, they may not have to suffer advertising. That's not good news for Google, but it's very good news to everyone else. The smart machine era will be liberating in ways we cannot imagine today.

Impact on Employment

So, what will people do? That's a great question, which is of course unanswerable. Even if you knew the answer, nobody would understand it. This is not new; if you went back in time to 1975, you could never explain why or how a service technician keeps Facebook running. The future will bring unknowable companies and jobs.



We do know one thing: the economic impact will be huge. Analysts differ on specific numbers and penetration rates. But, they all agree on the magnitude: trillions. The smart machine era will be a multi-trillion-dollar feast. Exploding economies generate many new jobs and new roles.

That certainly doesn't imply that you can ignore the disruption. This eBook explores ways that you can follow the advice from my son. We all need to embrace the robot overlords. Every company must seek a way to leverage the technology. My advice, simply put: find your link to things that think.

SOFTWARE IS EATING THE WORLD

Software is rapidly becoming the core of all industries. Most industries are unprepared. The change goes much deeper than adopting a new technology. The fundamental change is a dramatic shift in the basis of competition. This shift is cultural. The shift will disrupt executive composition. The shift will upset decades-long supply-chain relationships. Famous Silicon Valley VC Andreessen Horowitz says it best, "Software is eating the world."

The current scramble to implement autonomy for cars is a great example. Since the inception of the industry, automotive manufacturers competed on the basis of mechanical design and style. Suddenly, the industry sees that autonomy is a huge threat to that competitive basis. Future cars must implement autonomy to compete.

The supply chain is little help. While modern cars have millions of lines of code, they come mostly embedded in the myriad devices and "ECUs" in the vehicle. Integration between these devices is intentionally as simple as possible. For instance, a "smart cruise controller" may include a radar and speed algorithm. But, it doesn't expose the radar data itself, making it difficult to fuse its readings with video cameras or LiDARs.

The OEMs (car brands) realize this. Most are spinning up software teams to tackle sensor fusion, situational awareness, planning and more. However, most executives in the industry come from mechanical engineering or styling backgrounds. They are ill-prepared to hire and manage large coordinated software teams.

The auto industry is hardly alone. Future medical devices must work together to take better care of patients. The power grid needs distributed computing and control that can integrate fast renewable sources. Oil drilling platforms will be robotic systems connected to the cloud. Process control plants will leverage AI components that optimize every operational factor. In every industry, smart distributed software will make or break competitive positions.

These companies are expert at making cars, locomotives, power generators and medical imaging systems. How could these, how will these, turn into software companies?

But turn they will, because turn they must. Any company that ignores this disruption or delays entry too long risks losing their entire position to smarter competitors or entrants.

John Chambers, Cisco's legendary CEO, said, "You don't compete against competitors. You compete against market transitions." The smart machine era is the mother of all market transitions. The transition is towards a future dominated by software and intelligence expertise. It will impact every industry that has a physical presence in the real world. The Internet of Things turns out to be well-named. It will connect everything and thereby, change everything.

> *"You don't compete against competitors. You compete against market transitions."*
>
> **John Chambers**
> **CEO, CISCO**

WHAT IS THE INDUSTRIAL IOT, ANYWAY?

Clarifying the terms of the IoT

The term "Internet of Things" has become confusing. It combines many technologies, encompasses vast use cases, and attracts vendors of all sorts who want to join the fray. The IoT industry offers a jumble of confusing terms. There's little clarity across the industry, even for some of the most common terms. Worse, definitions from experts rarely help most readers. For instance, the IIC Vocabulary document defines "cloud computing" as a "paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on-demand." That mouthful does little to resolve confusion for most non-IT experts. For experts, I recommend the IIC's Vocabulary document. For the rest of us, Table 1 presents very simplified, yet quite usable definitions of the key terms used in this document.

Three Great Spheres of the IoT

The IoT is huge, but there are similarities between applications. Broadly speaking, there are three great categories, or "spheres," of applications in the IoT. These cross traditional industry segments. It is therefore much easier to understand the IoT by looking at the technological goals of applications rather than verticals like manufacturing or transportation.

The three spheres are:

- **Device Monitoring**: Device monitoring IoT applications simply connect a device to a cloud service. They are similar to a smartphone app, except instead of a phone, the end entity is a "thing." From a connectivity perspective, each thing usually has only a single connection to an IoT platform. This category includes most of the "consumer" IoT including thermostats, wearables and smart-home locks.

Table 1. Easy Glossary

Cloud: A huge array of computers in a multi-billion-dollar data center.

IT (Information Technology): Software (and hardware) that runs in the cloud. Also, sys admins and business programmers who run the above.

Things: Any physical device or system that has computing. Soon, everything manmade.

OT (Operational Technology): Computing that actually controls and powers "things." Also, engineers who implement the things.

Edge: IT's collapsed view of the (entire) real world not in the cloud.

Fog: Computing that makes OT things intelligent with distributed or elastic computing not in a data center.

IoT: All of the above.

Industrial IoT (**IIoT**): IoT that consumers don't buy (but including autonomous cars).

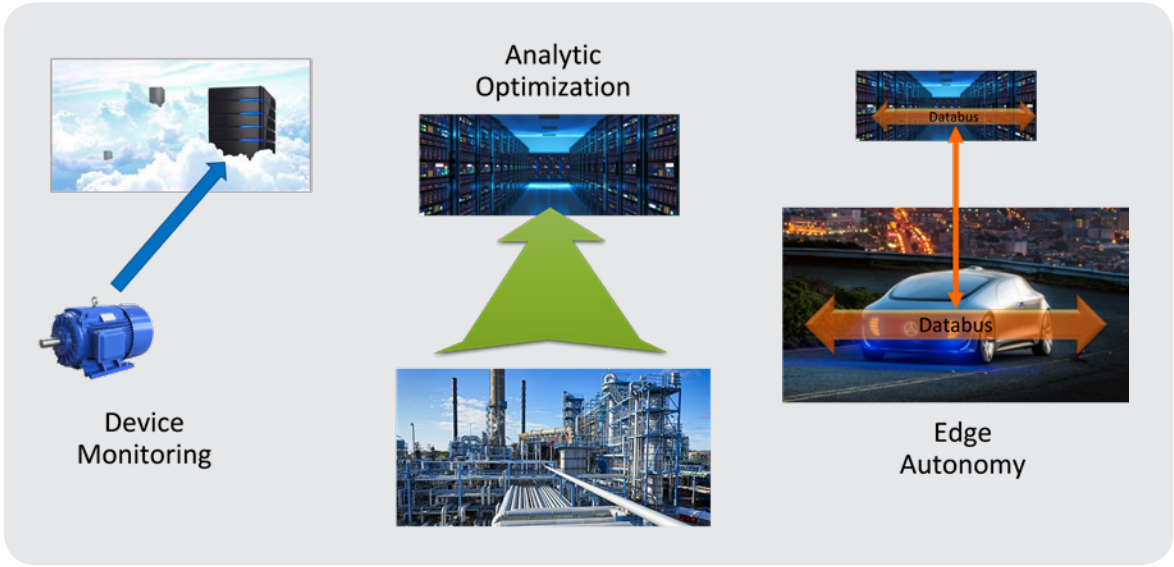


Figure 4. IoT Spheres.
The three major categories of applications in the IoT. Device monitoring applications connect a single device to a single cloud service. Analytic optimization applications collect huge datasets for cloud analysis to improve operations. Autonomy applications seek to build a new infrastructure by connecting many devices in the field. The first two categories focus on improving existing systems. Most autonomy applications target new application spaces.

The most important industrial monitoring applications are "predictive maintenance" systems that upload device data to a cloud service. The cloud systems then analyze the data to predict when a part may fail, allowing proactive repair.

- **Analytic Optimization**: Many IoT applications seek to optimize current system designs and operations. Examples include power plants, oil pipelines, factories and various industrial machines. The "things" here are sensors and actuators that work together as industrial systems. The fundamental goal is to get data for system-level analysis, often in the cloud. The analytics provide clarity that allows tuning the plant or its value chain.

  From a connectivity perspective, most optimization applications use the collect → gateway → IoT platform → analyze → optimize pattern. While all use cases unite field and cloud, this use case is usually what people talk about when they cite "IT/OT convergence," because it merges both existing historic computing realms. The "collect" step may involve many related devices. Typically, there's little to no connection between devices. Data flow is mostly one-way from the devices to the analytics. The feedback to the plant may only be through a human operator. Optimization applications thus connect many related devices in a plant, usually to a single destination.

- **Edge Autonomy**: IoT autonomy applications build new architectures for future connected infrastructure. Most of these applications are really building new things, although there are a few radical retrofits. Applications include autonomous transportation systems, smart medical systems with “doctor in the box” intelligence and smart future coordinated power grids. The real driver in this category is the magic of connecting functional artificial intelligence (AI) with pervasive networking. Connectivity is truly challenging—linking many devices in the field, fog and cloud. Autonomy applications can grow very large, even combining systems into supersystems. In these applications, the cloud is just one component of the connectivity and computing infrastructure.

Market Evolution

The analytics enabled by optimization and monitoring are financially compelling for reducing current operating expenses (opex). Thus, these early applications are why conservative plant managers are thinking of connecting their operations to the Internet. Unfortunately, far too many people think analytics are the entire Industrial IoT. We consider it only the first killer application. This is similar to the first killer application that drove the original Internet—email. Email was immediately compelling to existing applications.

Long term, however, both email and analytics represent only a small portion of the value of pervasive connectivity. Immediate opex reduction is clearly compelling. However, the future trillion-dollar impact of the IIoT will mostly result from bold new designs. Just as web applications are now more important than email, the stunning impact of new, autonomous infrastructure will eventually overshadow optimizing our current infrastructure.

A PRACTICAL GUIDE TO IIOT CONNECTIVITY

Connectivity is perhaps the most important aspect of the IIoT. But since the IIoT is so vast, there are many connectivity options. The Industrial Internet Consortium's (IIC) primary mission is to resolve the resulting confusion and thus guide the industry. The IIC developed the Industrial Internet Connectivity Framework (IICF) to help designers understand the many standards and choose the right one for their applications.

The IICF focuses on the layers above the network packet exchange. It offers profound insights into architecture, standards and use cases. It also presents a way to build an expansive IIoT network in the future by linking a few "core connectivity standards" that address different regions of the connectivity space.

The IICF is a very deep analysis. It defines a new "IIoT connectivity stack." It sorts the various standards into the stack based on their provided interoperability. It offers deep analysis of the six key standards and technologies with the most traction: DDS, OPC UA, oneM2M, RESTful HTTP, MQTT and CoAP. It also presents a grand architecture to combine these standards into more complex systems and a future truly interconnected Internet with multiple standards.

But the first goal of most designers is to get started in the right direction. The IICF analysis found that the IIoT space is so big that the connectivity technologies essentially do not overlap. Thus, by understanding the use cases, architectures and target end users, it is possible to select a best-candidate connectivity standard for most problems.

So, at the end of this section, we will use the IICF results to present a simple set of yes-or-no questions. Answering these questions should make it easy for you to assess your applications' fit to each technology.

Connectivity Technologies Don't Overlap

The IICF contains the deep insights of many experts, including those from the top industry consortia, many companies and most of the important standards. Its most surprising conclusion: the IIoT is so big that the technologies don't really overlap. The impression that there is overlap is mostly just confusion.

Designers may think they can choose any standard and succeed. But this implies that the IIoT connectivity solution space overlaps, as in the Misconception figure.

The reality is very different. The IIoT covers many industries with very different use cases. The range is breathtaking. There are thousands of companies and uncountable thousands of applications in the IIoT space. The IIoT really is the technological future of the entire world.

The connectivity technologies and standards that target these applications are very different. In fact, the IIoT space is so big that the technology options barely overlap. Today's architecture challenge in the IIoT space is therefore not one of choosing among overlapping standards that may each be able to reasonably solve a problem. The challenge is understanding the technologies, comparing the intended use to the application, and choosing the one that best addresses the particular challenge. Sure, stretching a technology all out of proportion may make anything work. But, that will result in a lot of extra work and an awkward design. If you look at a more realistic map of the situation, it looks more like the sparse Venn diagram in the Reality figure than the overlapping one in the Misconception Figure 5.

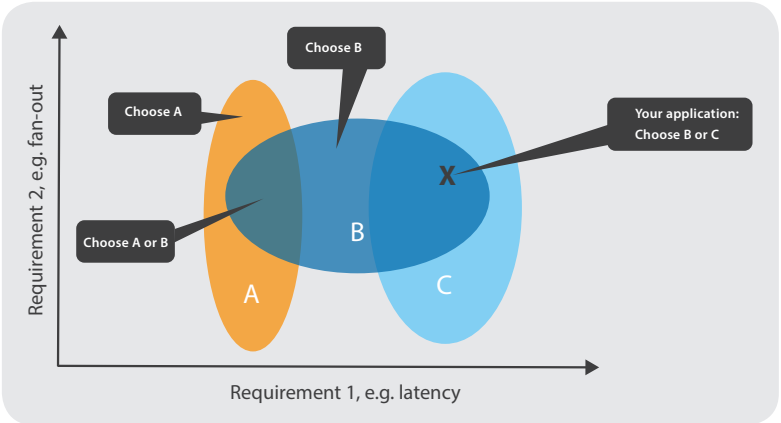


Figure 5. Misconception. Before the IICF, people (including the author) assumed that competing standards meet overlapping requirements in the IIoT connectivity space. This designer's application (X marks the spot), fits B or C; either should work.

This conclusion can be nonintuitive. After all, connectivity technologies all move data, right? They are nonetheless very different. In fact, one of the biggest red flags we see is when someone wants to create a "connectivity wrapper" that can abstract these technologies. Typically, they do

this to defer a decision until later. The wrapper lets them think "any protocol will work." In our experience, that never works out.

How can that be, really? The "connectivity wrapper" fails for the same reason that you can't pedal a train. Cars, trains, bicycles and feet all move people. And, if you're discussing alternative ways to commute to work, they may even be viable options. But if you want to move tons of freight to New York, or you want to go upstairs in your house, they are suddenly very different indeed. For these use cases, you simply cannot defer the transportation option selection. You must figure out your needs and then select the right capability.

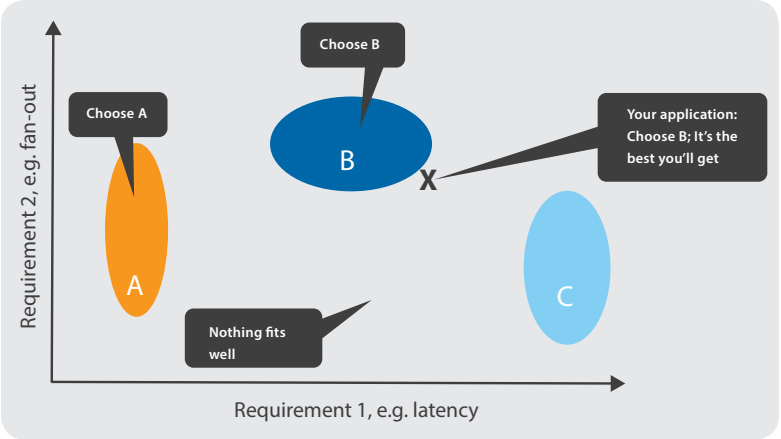


Figure 6. Reality. It turns out, the connectivity standards do not overlap. Most applications will not be a perfect fit and must adapt. Your challenge is sometimes choosing something imperfect and making it work.

This is a generic truism. Every abstraction has gains and losses. Abstraction that simplifies something for a specific use case is useful. Abstractions between similar things add flexibility. But, an abstraction that tries to pretend dissimilar things are similar sacrifices the power of each option, leaving only the "least common denominator" use cases.

Since the connectivity options are so different, in most use cases, there is no choice in connectivity technology. This may sound distressing, but in reality, the lack of overlap in the IIoT space actually makes an architect's task much simpler. The real problem isn't choosing between similar options; it is understanding the different options and overcoming biases.

The Industrial Connectivity Stack

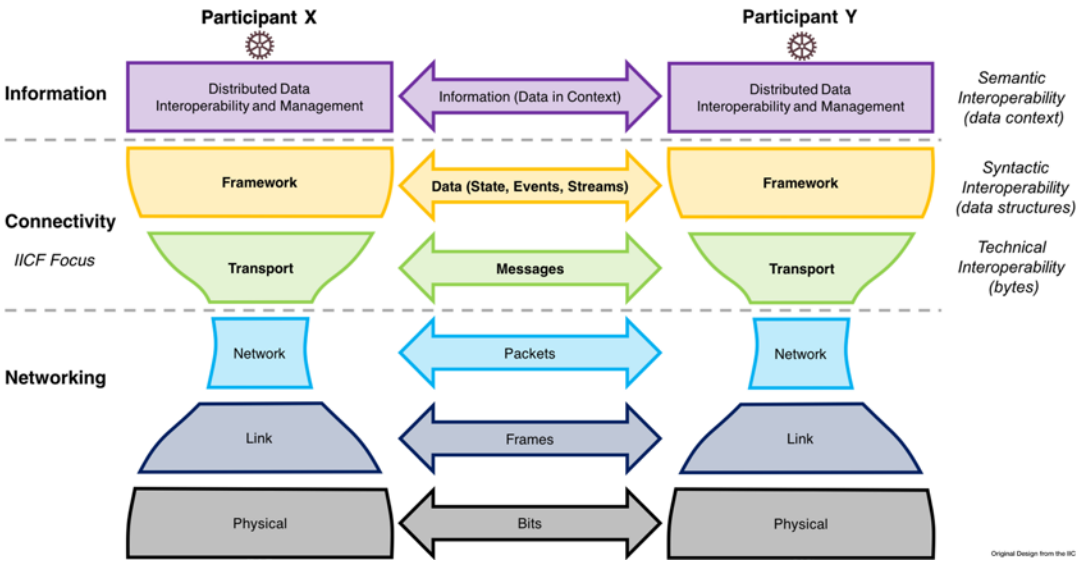


Figure 7. IIoT Connectivity Stack. Below layer 3 "Networking," the IIoT stack duplicates traditional Internet stacks. Above, it focuses on clarifying the interoperability that industrial IoT systems need.

The Internet has well-developed layered stack models, most notably the OSI "7-layer" model and the Internet "4-layer" model.

However, the IIC experts found that these models did not adequately capture the requirements of industrial networks. Thus, the IICF defines a new model, called the IIoT Connectivity Stack Model.

Both the Enterprise Internet and the Industrial Internet function by sharing data. However, industrial systems present different challenges than enterprise networks do. Most importantly, industrial systems combine complex, intimately interconnected software modules and devices. Interoperability between the various components is the most demanding requirement on the architecture. With crisp definitions of interoperability, the other requirements can be much better understood. The IIoT Connectivity stack shares layers 1-3 with the OSI model and defines levels 4 through 6 in terms of the interoperability provided.

The layers focus on what is exchanged. At the network layer, participants exchange bounded-length packets of information. This is usually implemented by the familiar Internet Protocol (IP).

Above it, the transport layer exchanges variable-length messages. Participants share opaque sequences of bytes. Participants using this layer can share information, but

interpretation of that information is completely up to the applications. This enables basic communications, but makes it difficult to interoperate between devices and software that are not designed together.

The next layer, called the framework layer, adds structure to the data exchange. This allows components to understand how to process the messages, also called "syntactic" interoperability. Participants above this level can use different programming languages, operating systems and processor architectures transparently. The framework layer also enables configurable quality-of-services (QoS) like reliability, durability, filtering and more. QoS enables control over data delivery, including selecting information and endpoint delivery conditions. Together, these functions enable a "data model" for the system. The data model is the basis for diverse components to work together. Sophisticated implementations can even match some differences in data model, thus allowing a large distributed system to grow incrementally from parts that are not all developed or deployed together. Thus, syntactic interoperability is critical functionality for an industrial Internet.

Once participants can exchange known structures, they must also know how to interpret the information, aka "semantic interoperability." This is the responsibility of the distributed data interoperability and management layer. In the current state-of-the-art, semantic definition is only practical within an industry. There are many standards that operate at this level, including the ICE (Integrated Clinical Environment) in the medical industry, Open FMB (Open Field Message Bus) in the power industry, and others. Semantic interoperability is beyond the scope of the IICF.

Combining Standards: The Core Connectivity Standard Architecture

The IIoT space is far too big to expect a single connectivity standard to span everything. Thus, to build an Internet, we will eventually need to connect subsystems based on different standards.

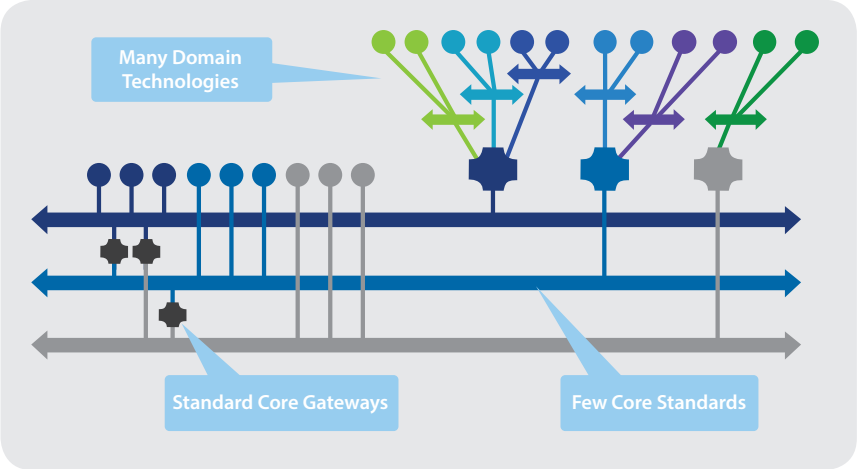


Figure 8. Core Connectivity Architecture. Each core connectivity standard requires a standardized core gateway to connect to the other core architectures. Other domain technologies can then interface to the system via any core standard. This design scales linearly with the number of technologies, thus enabling a true Internet.

The IICF does this with the concept of a "Core Connectivity Standard" (CCS). The CCS design eliminates the "N-squared" problem by choosing a few standards that together span the space and separately provide key functionality. The design simply defines the few standard bridges (called "core gateways") between core standards. Other connectivity technologies can then interface to the system through any one CCS. This enables practical end-to-end data exchange, as shown in the Core Connectivity Architecture figure.

This design enables a scalable, deeply connected future Industrial Internet of Things. Of course, it does introduce the question of what qualifies a standard to be a core standard. The IIC reasonably requires that a connectivity core standard shall:

- Provide syntactic interoperability
- Be an open standard with strong independent, international governance, and with support for certifying or validating or testing interoperability of implementations
- Be horizontal and neutral in its applicability across industries
- Be stable and deployed across multiple vertical industries
- Have standards-defined Core Gateways to all other connectivity core standards

With that definition, the IIC experts proceeded to define criteria, survey standards and evaluate all of the standards against those criteria. The criteria definitions alone are a huge contribution, beyond the scope of this paper. They resulted in an "Assessment Template" with deep analyses of the six standards with the greatest IIoT traction: DDS, OPC UA, oneM2M, RESTful HTTP, MQTT and CoAP. The assessment includes business, usage, functional and implementation viewpoints. It is a unique analysis of connectivity technologies for industrial systems. In the end, the IICF identified four of the six as potential core standards: DDS, OPC UA, oneM2M and RESTful HTTP.

Of course, this design targets a future world of vastly connected systems. There is a more immediate question: "if I'm starting an IIoT project, what should I use?" Next, we present the IICF guidance and a simple tool to navigate that guidance.

Choosing a Connectivity Standard

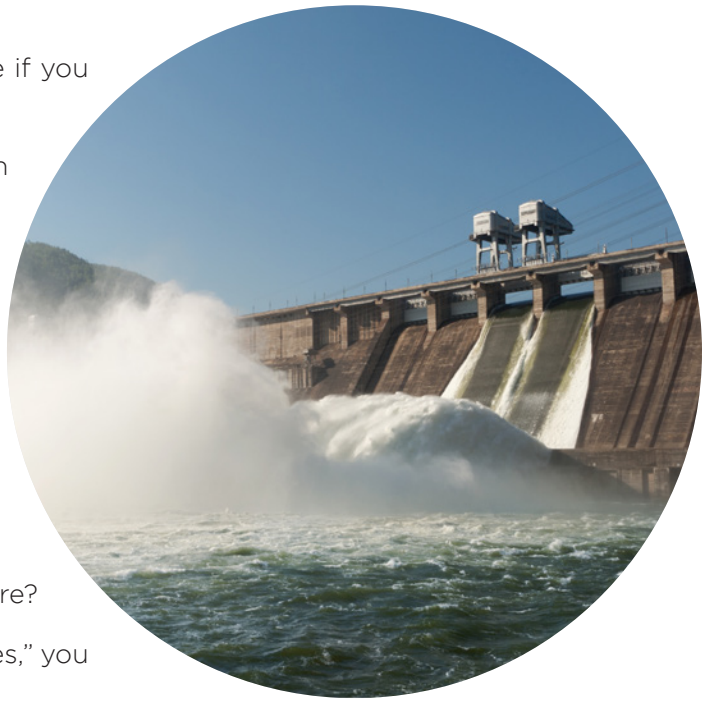
In one key table, the IICF outlines the example use cases and application spaces of the various core connectivity options. The intention, and the effect, is to provide useful guidance to architects struggling with the confusing arrays of vendor and advocate claims. The table is reproduced below in Table 2.

Table 2. (IICF Table 8-2) Non-overlapping system aspect examples addressed by the potential IIoT connectivity core standards. The IIoT space is so big that the primary users of the different technologies don't speak the same language. In practice, there are few applications that could reasonably make alternative choices. The real problem in the industry is the lack of understanding of the options. The IIC experts sorted the technologies into very clearly different "boxes." The choice is nearly as simple as identifying which box most sounds like you.

System Aspect	Example User	Approach	Targeting Standard
Software Integration and Autonomy	You are a software architect. You are building a system or product line, and you control the architecture. You critcally need to integrate components written by different programmers or even entire teams.	A data-centric approach will define the interfaces, capture the dataflow, enable module evolution, and enforce interoperation between teams. This approach also eases redundancy, fast complex data flow, and selective data filtering.	DDS
Device Interchangeability	You are a device manufacturer, with the goal of making devices that will sell into many applications. The device offers services, such as configure, start, stop, etc. You have no idea how the device will eventually be used. Your users are likely not software experts; they just want to add or integrate the device into a work cell.	A device-centric approach will allow the device users to write generic software that will interoperate with competitor's devices.	OPC UA
Web and Mobile User Interfaces	You are building mobile apps or web browser based applications to provide the human machine interface. You need an easy way to support clean human integration and access to backend services.	A RESTful approach will make it easy to connect to many types of enterprise systems and UI devices.	Web Services (HTTP/REST)
Information & Communication Technology (ICT) Integration	You are building a wide-area wireless system that needs to allow applications and devices to share data and information. The devices use various technology and domain-specific protocols. The application and devices you integrate rely on leveraging the services provided by the communications provider network.	A common, standard services-layer approach enbles applications and device to share data and information without forcing the application to understand mulitple protocols implemented on the devices. The applications can thus run in the Platform Tier and seamlessly connect to diverse IoT devices in the field.	oneM2M

Let's take this process a bit further. It's possible to ask a few very simple questions for each technology option and quickly narrow the choices.

DDS



Here are five questions to answer and determine if you need DDS:

1. Is it a big problem if your system goes down for a short time?
2. Are milliseconds important in your communications?
3. Do you have more than 10 software engineers?
4. Are you sending data to many places, as opposed to just one (like to the cloud or a database)?
5. Are you implementing a new IIoT architecture?

If you answered three out of the five questions "yes," you probably should use DDS.

DDS is a series of standards managed by the OMG that define a databus. A databus is data-centric information flow control. It's a similar concept to a database, which is data-centric information storage. The key difference: a database searches old information by relating properties of stored data. A databus finds future information by filtering properties of the incoming data. Both understand the data contents and let applications act directly on and through the data rather than with each other. Applications using a database or a databus do not have a direct relationship with peer applications.

The databus uses knowledge of the structure, contents and demands on data to manage dataflow. It can, for instance, resolve redundancy to support multiple sources, sinks and networks. The databus can control Quality of Service (QoS) like update rate, reliability and guaranteed notification of data liveliness. It can look at the data inside the updates and optimize how to send them, or decide not to send them at all. It also can discover and secure data flows dynamically. All of these things define interaction between software modules. The data-centric paradigm thus enables software integration.

So how does this satisfy the five questions?

1. Since it is directly controlling flow, a databus does not require servers. So, there's no single point of failure. The downtime required to reboot a server and remake connections unexpectedly is never necessary. Without direct relationships with peers, redundancy is transparent. If the application is managing a thermostat, optimizing a plant, or assembling parts, a short downtime is not catastrophic. But, if the software is responsible for someone's breathing or the stability of the power grid, even short interruptions cannot be tolerated.
2. Since the databus has full control over how data flows, it can send information directly between peers. Thus, it can deliver in times measured in milliseconds or microseconds. DDS can use multicast intelligently when available. It knows delivery deadline requirements and can measure if the system is meeting delivery times. So, it can warn applications if the network (or anything else) can't handle the needed flow rates.
3. Teams of programmers must control interfaces between modules. The databus specifies a full data model. All connectivity frameworks do this to some extent, but the databus specification is more expressive. It includes not only type information, but also QoS, such as deadlines, sensor availability and flow rates. So the interfaces are defined and then enforced at runtime. The databus can also manage the evolution of those interfaces allowing modules, for instance, that use newer and older versions of an interface to interoperate. That's important for a practical large IIoT system that must be incrementally deployed and updated.
4. A databus controls flow between many complex applications. It handles a mix of fast and slow components. Its filtering can make the overall flow manageable. Peer discovery delivers data between multiple field-based components. And QoS control guarantees the flows. There are simpler solutions for one-way, one-destination flows like capturing sensor information to send it to the cloud for analysis.







5. Finally, using a databus is a completely new approach, not an incremental design change. Most DDS designs are building something new rather than optimizing something old. That said, many databus designs integrate legacy subsystems via gateways. And, systems with large installed infrastructure often implement a migration strategy with smart adapters that put components on the bus, allowing them to go through their normal depreciation cycle.

Most databus systems don't have all five of these properties. Three of the five indicate that a databus design will be compelling.

OPC UA

OPC UA is a standard managed by the OPC Foundation, also documented as IEC 62541.



OPC UA targets device interoperability. Rather than accessing devices directly through proprietary application program interfaces (APIs), OPC UA defines standard APIs that allow changing device types or vendors. This also lets higher-level applications, such as human-machine interfaces (HMI) find, connect to and control the various devices in factories.

OPC UA divides system software into clients and servers. The servers usually reside on a device or higher-level Programmable Logic Controller (PLC). They provide a way to access the device through a standard "device model." There are standard device models for dozens of types of devices from sensors to feedback controllers. Each manufacturer is responsible for providing the server that maps the generic device model to its particular device. The servers expose a standardized object-oriented, remotely-callable API that implements the device model.

Clients can connect to a device and call functions using the generic device model. Thus, client software is

independent of the actual device internals, and factory integrators are free to switch manufacturers or models as needed. So, OPC UA can build and maintain a system from interchangeable parts, much like standardized printer drivers allow PC system integration. Note that the device model also provides a level of "semantic" interoperability, because the device model defines the generic object APIs in known units and specified reference points.

Determine if you should use OPC UA by answering the following questions:

1. Are you in discrete manufacturing?
2. Are you associated with the German Plattform Industrie 4.0 program?
3. Are you building a device that will be integrated by control or process engineers or technicians, rather than software engineers?
4. Will your product be used in different applications in different systems, as opposed to one (type of) system where you control the architecture?
5. Are you building equipment for a "workcell"?

Three "yes" answers to these five questions point strongly to OPC UA. Let's look at how the technology fits these use case indicators:

1. OPC UA is well-positioned for discrete manufacturing. These applications are characterized by integrations of devices into sets of tightly coordinated subsystems. Since users want to avoid vendor lock-in, a ready supply of devices with interchangeable device models is important.
2. The German initiative Plattform Industrie 4.0 recommends OPC UA. Industrie 4.0 is focused on manufacturing, in contrast to the IIC that works on IIoT technical system architectures across verticals. In a sound bite, Industrie 4.0 is about making things, while the IIC is about making things work. They intersect only in manufacturing systems.
3. OPC UA provides more than connectivity; it also has pre-defined device models and a device integration architecture. Those using and choosing it usually target users who are plant or process engineers, rather than programmers. System integration in

manufacturing is usually done between devices, not software modules. OPC UA has very helpful device models that aid interoperability between device manufacturers.

4. OPC UA has a system discovery mechanism called an "address space." It builds an object hierarchy covering all the devices and subsystems at runtime. This can be rolled up to a site-wide server, for instance, and the system connected to a site HMI. This dynamic system building is very useful for providing similar functionality, such as historian storage or HMI viewing, to very different applications. It is appropriate where your users control the system design, not you.
5. Most OPC UA systems end up in workcells. These are stand-alone subsystems, usually incorporating 20 devices or so. OPC UA, and especially the industrial-integration software that supports it, targets workcell integration. The address model and object-oriented nature directly support a hierarchy of these workcells. Users of the other standards rarely characterize their use cases as "workcells."



ONEM2M

OneM2M provides a common service layer that sits between applications and connectivity transport. Its emphasis is on providing common services, on top of different connectivity standards.

To determine if you should use oneM2M, consider these questions:

1. Do you know what "ICT" stands for, and does it describe what you do?
2. Is the cellular network your primary connection technology?
3. Are your target applications largely composed of moving parts?
4. Can the components of the system tolerate intermittent connections and loosely-controlled latencies?
5. Will the system leverage services provided by a communications provider, such as a telco?

These questions differ in character from the questions about the previous technologies. OneM2M results from cooperation among many mobile wireless providers. It targets networks of mobile devices that communicate mostly or only through the base-station infrastructure.

The following points examine why oneM2M is implied by these questions:

1. Surprisingly, most target users of DDS and OPC UA cannot even correctly define the name of oneM2M's target industry as Information and Communication Technology (ICT). There are, of course, exceptions. But, if you consider yourself in the ICT industry, then you need to consider OneM2M; it was designed for you.
2. The core design of oneM2M is to define services that mobile devices can use to cooperate and integrate. If you are going to use those services, you need to connect to them. They will be running in the platform layer (cloud) connected mostly through the cellular data infrastructure. Other technologies also use IP traffic over the cell network, but they usually also heavily leverage LAN, local wireless or WAN networking technologies in their designs.
3. There is a potential future market for 5G wireless integration of fixed assets, such as manufacturing cells. However, this technology is still years away. oneM2M performs best for mobile assets. One especially powerful aspect is that oneM2M abstracts differences in protocols to those devices. Thus, it can integrate different ways to connect to similar devices.
4. Cellular mobile systems are not reliably connected. Thus, applications must not fail when communications are offline for a few seconds or minutes.
5. OneM2M system designers assume a cloud backend in their designs. The core of oneM2M is the standard services layer that is provided by a telco or its partners.

RESTFUL HTTP



REST (Representational State Transfer) over HTTP is the most common interface between consumer applications and web services. REST is an architectural pattern for accessing and modifying an object or resource. One server usually controls the object; others request a "representation" and may then send requests to create, modify or delete the object.

To see if RESTful HTTP is the best candidate for your application, ask these questions:

1. Are you connecting independent devices to a single web service API?
2. Are you building an HMI interface to an IoT device or service?
3. Does your application only need to be fast enough for human interaction?
4. Must your dataflow cross firewalls that you do not control?
5. Is there no device-to-device communication?

Three "yes" answers indicate you will likely be best off with RESTful HTTP. The reasons:

1. RESTful HTTP fundamentally makes it easy to connect a field device to a web service. REST is the most widespread way to build web services, enabling copious offerings to help developers. For instance, the DreamFactory open source project automatically creates APIs from most any database, thus enabling a centralized data-centric approach.
. While most applications use hypertext to present a linked view of a web page to a user, this paradigm is also similar for many IIoT "monitoring" applications. These applications are similar to smartphone apps, except there's no human user. Instead of a phone, the end entity is a "thing," typically a single device. From a connectivity perspective, the things usually have only a single connection to an IoT platform. This category includes most of the "consumer" IoT, including thermostats, wearables and smart home locks.

. The most important industrial applications are "predictive maintenance" systems that upload device data to the cloud. Cloud systems then analyze the data to predict when a part may fail, allowing proactive repair.

2. HTTP is the most common way to serve information to human users. While most think of HTTP servers as living in the cloud with clients on the edge, this often is not the case in the IoT. The server often in fact runs on the device itself, providing an intuitive configuration and management interface. Browsers can simply connect directly to the device, giving full access to advanced visual capabilities. This is limited to local connections, so another common configuration is to combine this with the above by having both the device and the user connect to a cloud-based HTTP service. This allows easy remote access to the device.



3. RESTful HTTP is an approach, not a standard with an official type system. However, most web systems use text encodings like JSON or XML. Text encoding is not fast so response speeds may satisfy humans, but not fast machines. Binary systems like Google Protocol Buffers (Protobuf) and the new HTTP/2 standard offer more efficiency, but are not yet widely used. HTTP runs over TCP, which does not deliver low latency.

4. Because all websites offer HTTP services, most IT firewalls allow connections to HTTP's default port (80). Deep packet inspection systems will likely accept HTTP. Thus, using HTTP on port 80 is usually the easiest way to traverse firewalls without special configuration.

5. RESTful HTTP requires that all devices connect to a server. While multiple devices may interact through that server or share a database, applications that share data between devices are rare.

MQTT

MQTT is a very simple protocol designed mostly for the "data collection" use case. It does not qualify as a "core connectivity standard" per the IICF guidelines, because it has no standard type system. Without a type system, it cannot offer a standard ability to interoperate at the "syntactic" data-structure level, leaving all data interpretation to the application.



Nonetheless, there is significant awareness around MQTT. Because of its simplicity, simple questions about your system will help determine if it is appropriate:

1. Do you think of your application as data collection?
2. Is there little device-to-device communications?
3. Is interoperability not a consideration?
4. Do you have many small devices?
5. Is software a minor challenge?

Again, if you answer three of these "yes," you should look at MQTT. The reasons:

1. The first "T" in MQTT stands for "Telemetry," or data collection at a distance. This is its main use case.
2. MQTT is designed as a hub-and-spoke design that requires a broker. It doesn't support direct inter-device communications. Choosing MQTT for device-to-device communications is awkward.
3. Without a type system, MQTT applications are tightly coupled–they must all be aware of the data format. The only way to build interoperability is to implement a way to share types in user code. As a result, most all MQTT applications are standalone systems. MQTT also does not help with system evolution; version compatibility between components must be provided by the applications.
4. MQTT is by far the simplest technology considered here. If you have many small devices that are simply connected, simple software can handle your challenge.

5. MQTT offers little to ease software development. There is only one QoS setting (reliability). There are no defined services. It offers no data or device modeling. You are therefore going to have to write all the software from scratch; that's only practical if you have a simple software challenge.



COAP

The Constrained Application Protocol (CoAP) is a connectivity transport standard inspired by HTTP, but designed to be more lightweight and efficient. It was established using the UDP transport. CoAP is designed to interoperate with HTTP and the RESTful web services through simple proxies. It targets low-power, wireless devices with few resources.

Like MQTT, CoAP lacks a type system and thus does not qualify as a Core Connectivity Standard.

To see if CoAP will work for you, answer these questions:

1. Are you running on a low-powered device?
2. Does your device have limited ROM/RAM?
3. Does your network have high package error rates and need throughput in a range of 10-100 kbit/s?
4. Do you need direct machine to machine communication, but not real-time?
5. Is your goal to connect your devices to a RESTful HTTP system?

Three "yes" answers point to CoAP because:

1. More than the other options here, CoAP conserves communication overhead. Chatty protocols burn energy, so CoAP is good for conserving battery power.
2. CoAP is not as simple as MQTT, but it provides much more functionality. Thus, if its RESTful paradigm fits your application, you will save overall code space.
3. Although it normally runs over UDP, CoAP offers "confirmable" option for reliable communication over an unreliable medium. Most implementations

target low-power wireless specifications like 6LoWPAN. These sacrifice speed for power.

4. CoAP sends messages between endpoints peer-to-peer. However, the reliability protocol uses an exponential backoff strategy that's inappropriate for a real-time system.
5. The primary use-case is for applications that want to use HTTP-like communications in constrained (low memory, low bandwidth) environments. It is basically a lighter-weight protocol that preserves the same communication model as HTTP: request-reply and REST. Servers make resources available under a uniform resource identifier (URI), and clients access these resources using methods, such as GET, PUT, POST and DELETE. This simplifies connection to web services.

Comparisons

Comparing these technologies highlights the stark differences and non-overlapping nature of connectivity approaches.

For instance, OPC UA is object oriented (OO), while DDS is data centric. Those are diametric opposites. The object-oriented mantra is "encapsulate data, expose methods." Data centricity is all about exposing data, and there are no user-defined methods. The only methods are defined by the standard.

OPC UA targets final device-centric integration by plant engineers. It offers easy interoperability between devices from different vendors. By contrast, DDS targets final data-centric software integration by software teams. As intelligent software gains importance, DDS provides the global data abstraction and dataflow interface control that software teams need.

OneM2M and RESTful HTTP aim at connection from the edge to cloud services. Unlike DDS or OPC UA, neither is often used for device-device communications. They truly operate in entirely different spaces. OneM2M works by offering common services aimed at integrating mobile devices. None of the other technologies target this application.

Neither MQTT nor CoAP have a standard type system. Thus, they leave interoperability to the applications. You should consider these Connectivity Transports only if you control the interfaces between all devices and servers.

MQTT applications mostly target data collection from devices to a central store or analysis function. This is a rare application for either OPC UA or DDS, which work between devices.

Finally, while all the technologies have connections to web services, CoAP is the only one that directly implements device connectivity with a RESTful pattern. It ideally fits that use case.

Looking at the differences, it's pretty clear that these technologies simply do not compete in practice. However, the level of "confusional competition" is amazing. The various vendors and standards organizations, in general, do not help. Their positioning often uses similar words for vastly different concepts. Common terms like "publish subscribe" hide huge differences in types of information, discovery, selection of data and QoS control. "Real time" without specifying a time period like milliseconds or minutes is meaningless. And, the Internet of "Things" leaves a huge range of "things" to the user.

Hopefully, this guide can help resolve some of this confusion. Most applications fit one of these popular standards. If you answered "yes" to three questions for any technology above, you should start there.

Unfortunately, there are many applications that don't easily fit any of the standards. If you didn't have three "yes" answers, or if the questions point to more than one, you are in "interstellar space." Your best bet is likely to choose the closest match and expect to do some adapting.

The Future Connected IIoT



Combinations of the core standards will make great sense in the future. For instance, future complex software systems can use DDS, but access interchangeable OPC UA devices through a gateway. That design is powerful.

As detailed above, the IICF Core Connectivity architecture allows sharing data across connectivity technologies to allow this future, pervasive Industrial Internet. IIoT use cases are evolving from simple monitoring to optimization and finally to autonomy. These increasingly require more powerful integration.

Today's IIoT designs are relatively isolated within industries. Someday, there will also be cross-industry integrations, such as manufacturing systems integrated with transportation and power. More importantly, sophisticated autonomy software will reconfigure workcells, creating a bold new world for component device vendors. Wireless 5G systems will interoperate with freeway controllers and autonomous vehicles. 5G may even directly control factory devices, eliminating wiring in manufacturing.

However, designers should consider the vastness of the space. Today, there are few concrete needs to bridge the huge gaps between connectivity systems. That doesn't mean the industry isn't responding to the obvious need. Gateway standard efforts are active between all the core connectivity standards. And, a recent demonstration at an IIC testbed shows a bridge between DDS and OPC UA. Long term, these integrations will allow bigger systems combining technologies. For now, designers must understand the vast differences between connectivity standards and choose the one that best fits their problem space.

THEORY: SYSTEMS, SOFTWARE AND INTEGRATION

These comparisons are simple to use, but they lack depth. Let's dig a bit, using as examples the two most sophisticated industrial frameworks: DDS and OPC UA. This section is technically deep; the more practical-minded reader may wish to skip to the introduction to a databus or all the way to the autonomous drive example.

Computing systems are fundamentally defined by shared information. Dynamic systems, such as controllers are about moving information or dataflow.

Architectures are essentially abstractions, and we will use the terms interchangeably here. Without a software abstraction, you're writing assembly language. The abstraction determines what types of entities interact. It defines how components find, use, store and update information. Importantly, it also determines where the state lives...a critical decision. The more powerful and general the abstraction, the more powerful and general the software.

At its most profound level, a connectivity technology implements an abstraction to integrate components into a system. The abstraction provides a model for integration, enabling, and controlling both dataflow and execution flow. Since integration is so key to connectivity, we will define an "integration point" to include the types of components integrated, the interfaces between components and the types of users who effect the integration.

Since it defines which entities interact, the abstraction also determines coupling. Decoupling, in general, increases the flexibility of an architecture. It's no accident that poor software is often called "spaghetti code." Systems must consist of interacting parts, but every coupling increases complexity.

The fundamental differences between technologies stem from their different abstractions and integration points. Complexity and maintainability is largely a function of coupling. The architectural challenge, then, is to create a powerful abstraction to effect integration with minimal extraneous coupling.

OPC UA

OO SYSTEM ABSTRACTION

OPC UA offers a fundamentally OO abstraction. OO is an architecture that composes systems from objects that include both methods (execution) and state (data). Objects interact by calling each other's methods. OO encapsulates information effectively. It is simple and intuitive. Nearly every programmer is trained in OO design and implementation.

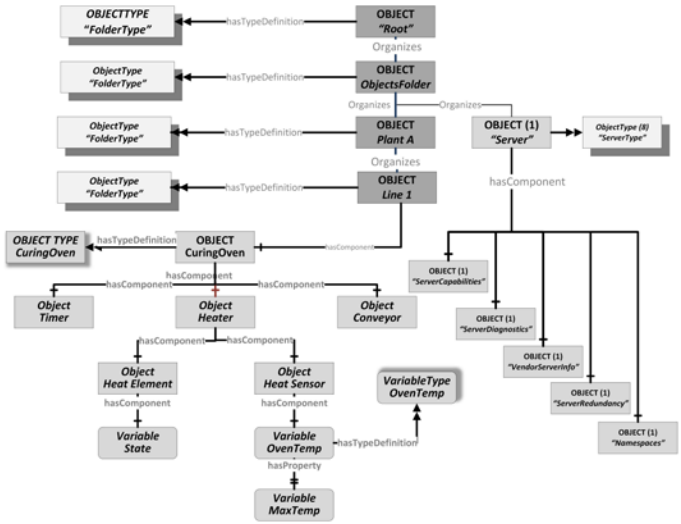


Figure 9. OPC UA Information Model Example. This diagram outlines how OPC UA would model a specific device, in this case a Curing Oven. Objects can be combined into hierarchies to define entire plants. At runtime, the system rolls up all the models in the plant and defines a browsable "address space" with live values for variables. (source: The Everyman's Guide to OPC UA by John S. Rinaldi)

OPC UA uses the OO abstraction to effect system information modeling. It defines an "Information Model" as "a representation of the concepts, relationships, constraints, rules and operations that capture the characteristics of an entity or process." The OPC UA Information Model contains both the data (operational state), and "meta data" like units and manufacturer part numbers. The model is a hierarchy of objects rolling up from devices to model the larger plant or system.

OO flow is fundamentally sequential. You "call a method" on a remote server/object, it returns, and then you call the next. In OPC UA, the paradigm is sequential reading and writing device values or streams. Data access traverses through multiple layers of servers until you get to the device. OPC UA added "pub/sub" functionality to take some advantage of parallel execution on a local network. But this functionality is a very simple add-on to the fundamental OO structure.

DEVICE INTEGRATION POINT

OPC UA integrates devices. Components are mostly devices, defined by standard device models. The models are very specific to a device. Applications may know, for example, that they are talking to a "starter motor" with start() and stop() methods, and even low-level information like current flow in windings. Since the models include units and shared meaning of these values, they enable semantic interoperability between devices from different vendors. Companion specifications define common devices across the industry, easing interoperability between devices from different vendors.

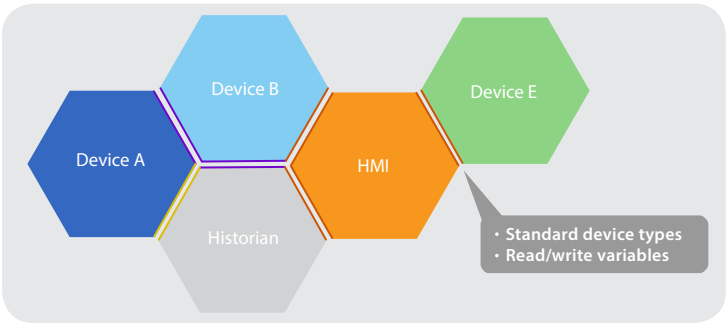


Figure 10. Device Integration. OPC UA integrates standard devices with an object-oriented architecture. It connects objects that model hierarchical systems of devices. Devices that conform to standard device models can be replaced by other vendor's parts. Objects interact with each other through defined methods and data.

Software systems built with OPC UA are typically compositions of devices and existing modules like historians and Human-Machine Interfaces (HMIs). Direct OPC UA users are mostly device vendors who implement the servers and device models on their products. End OPC UA integrators are typically control or process engineers building and configuring systems of devices and existing software modules.

At runtime, the OPC UA information model is implemented on servers as a browsable "address space". Clients can access parts of the system by traversing the runtime tree. HMIs and other software components can therefore access system data from this dynamic structure.

OBJECT COUPLING

OPC UA does a good job of decoupling device vendors. Since device integration is its main target, this is the source of most of its power. The UA community and process makes this even more useful; the OPC Foundation sets crisp standards for everything from protocol to device types, and then conducts conformance testing for vendors.

From a software standpoint, however, OPC UA uses a coupled object-centric model. Thus, both sides of the communication must be present simultaneously to perform the communications. A client, for instance, cannot invoke call without having a reference to the server, and

moreover having a handle to a concrete instance of that service. Thus, the objects are coupled in time (they must both be running) and space (they have to find each other). Less obviously, they are also coupled in flow, since the rate of flow is controlled by one end, the producer. This is true both for the main request-reply specification and the new publish-subscribe subsystem.

Objects create and consume data that has a "home" in a device or server. The state resides in servers, so high-availability execution requires complex state transfers between backups. The home must be active for the data to exist in the system.

DDS

GLOBAL DATA ABSTRACTION

Rather than an OO structure, DDS offers a shared "global data space" abstraction. The abstraction means that all data is available to every application when it needs it. Every application simply provides data it knows into the space and gets the data it wants from the space. It's naturally parallel; all the data is logically everywhere whenever anything needs it. It provides very useful caching, redundancy management, interface control and more. It's elegant and powerful.

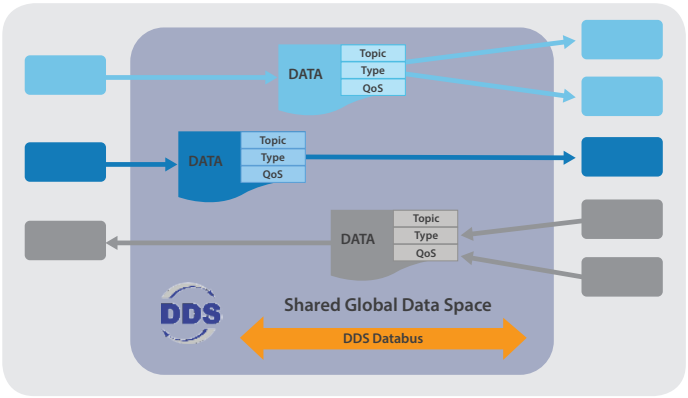


Figure 11. DDS Abstraction. The DDS shared global data space is a virtual concept. There is no actual global storage or server; the system instead provides the appearance that all data is everywhere. In actuality, the data lives in distributed caches on the various nodes. But entities have no view of that.

The "data model" interface control that combines schema (data types) and QoS (rules for use) is particularly useful because it directly models the dataflow. With dataflow control and understanding, it provides a global truth and rules for sharing that truth. This is data centricity, the same power that enabled databases to take over the enterprise. We'll examine it in more detail below.

Note that the DDS data model is relational, not hierarchical. Thus, participants locate data by searching for it by name or properties, just like a database. It hides the physical system structure. That makes it more flexible and easily scalable, at the cost of obscuring the mapping to the physical system.

Of course, data centricity introduces an unfamiliar paradigm, so the learning curve is steep. The concept is unique, the programmer's task very different, and the software has many knobs and dials. That's because DDS fundamentally moves much of the complexity from the applications to the middleware. But the end-user applications are typically much simpler overall.

SOFTWARE INTEGRATION POINT

DDS is fundamentally a software integration technology. It lets architects define a system data model, and then uses that to automatically enforce interfaces. It works well for building and integrating AI modules, custom software development and wide data distribution. Most DDS end users are teams of programmers with dozens or even thousands of developers.

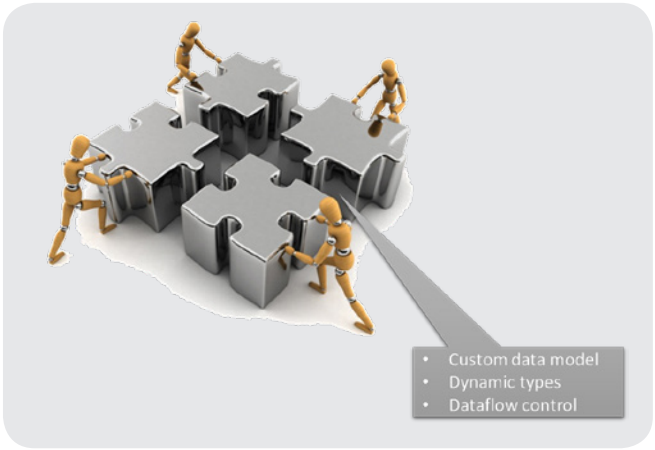


Figure 12. Software Integration. DDS integrates software modules through a common data model. It then delivers and controls flow through the data interfaces. Participants interact with the data model, not with each other.

Coding as a team requires, above all, clear interfaces. Clear interfaces imply a schema (what information is flowing) and a control model (how and when it flows). DDS provides both explicitly; in fact it a) allows you to define a data model (schema and flow,) and then b) enforces it. It even c) matches it for you as parts evolve separately. And it lets everyone work in their favorite language, CPU, transport, etc. The result is fully-parallel execution that supports multiple control models within the same overall structure. Each team member has a function to write, knows exactly the schema he or she must exchange, and then sets his/her QoS to ensure proper operation. It's fundamentally designed for team coordination. In fact, many users structure their teams by their DDS interfaces.

Interfaces between modules consist of subsets of the common system data model called "topics." Each topic includes a schema and data QoS. More importantly, when you provide or use data from the space, you tell the system your version of the schema and your QoS promise or needs. DDS uses that information to define and enforce dataflow between components, matching flow requirements and even resolving schema differences.

DDS started in autonomous systems at the Stanford Aerospace Robotics Lab. It is designed for Artificial Intelligence (AI) components. This may not mean human-like efforts like machine learning. But most of today's DDS systems include "functional AI," meaning smarts well beyond the current system. The DDS "all data everywhere" abstraction makes this easy; the AI specifies the dataflow and gets it (if possible.) AI systems with DDS are much more flexible.

AIs need data very differently than humans or fixed algorithms like controllers. Humans can intuit patterns in information, but mostly they need the ability to select a small amount of information at a slow rate. Controllers need fast information with little delay from known sensor sets. In both cases, designers can figure out what information is important before the system is running.

AIs are far more demanding. AIs sometimes need lots of data for analytics, select data for reactive behavior, and far-flung remote data for comparisons. They need to collect deep information for "training sets" without flooding the network. They can look for patterns within high-speed signals. Because they are so flexible and fast, it's hard to build a specific data design a priori. And it's getting even more challenging quickly; there will soon be many interacting AIs, both local and off prem.

DATA-CENTRIC DECOUPLING

In a data-centric system, the only coupling is to the data; data centricity is an architecture formed from participants that communicate only with the data infrastructure. Data-centric systems contrast with object-oriented systems (where objects communicate), message-oriented systems (where participants send messages to each other) and service-oriented architectures (where participants connect to services). Participants in data-centric systems are decoupled from all other participants in time (e.g. startup order), space (e.g. location) and flow (e.g. data rates). Examples include a database (data-centric storage) and a databus (data-centric communications). A data-centric distributed system (databus) enables scalable, real-time, dependable, high-performance and interoperable data exchanges.



Figure 13. Coupling. Data-centric systems decouple across time, space and flow.

Time decoupling means there is no dependency on startup or join sequence. Participants may come and go; adding or removing applications or flow paths doesn't affect others.

Space decoupling means data can come from any physical location. Producers and consumers may reside in devices, in the fog, or in the cloud transparently.

Flow decoupling means that dataflow rates or reliability specifications between endpoints do not interact. Each application can request data at a different update rate, over any network and with or without reliability guarantees.

The result is the opposite of spaghetti; all applications work in parallel without interference. Still, data-centric sharing lets them interact, thus letting them work together as one system.

The most common interaction pattern between participants and the data is publish-subscribe. But, publish subscribe is very different from data centricity. Publish subscribe is an interaction pattern that defines how entities interact. Data centricity is an architecture that defines which entities interact. DDS, for instance, supports publish-subscribe, request-reply and queuing interaction patterns. The key is that these interaction patterns are between a participant and the data, not between participants like clients or servers. Data-centric systems implement minimal coupling over any pattern.

Combining OPC UA and DDS

Of course, real systems include both devices and software. Therefore, while the primary need drives the choice of standard, it may not satisfy all the needs. Future OPC UA systems may compete on sophisticated software that needs DDS. Conversely, integrating legacy or replaceable devices into a DDS system may be easier with OPC UA. In these cases, the practical implication is you that need to design for multiple integration strategies.

This is where the IIC's Core Connectivity Standard design is so compelling. By combining the strengths of both standards, it explicitly accommodates future demands.

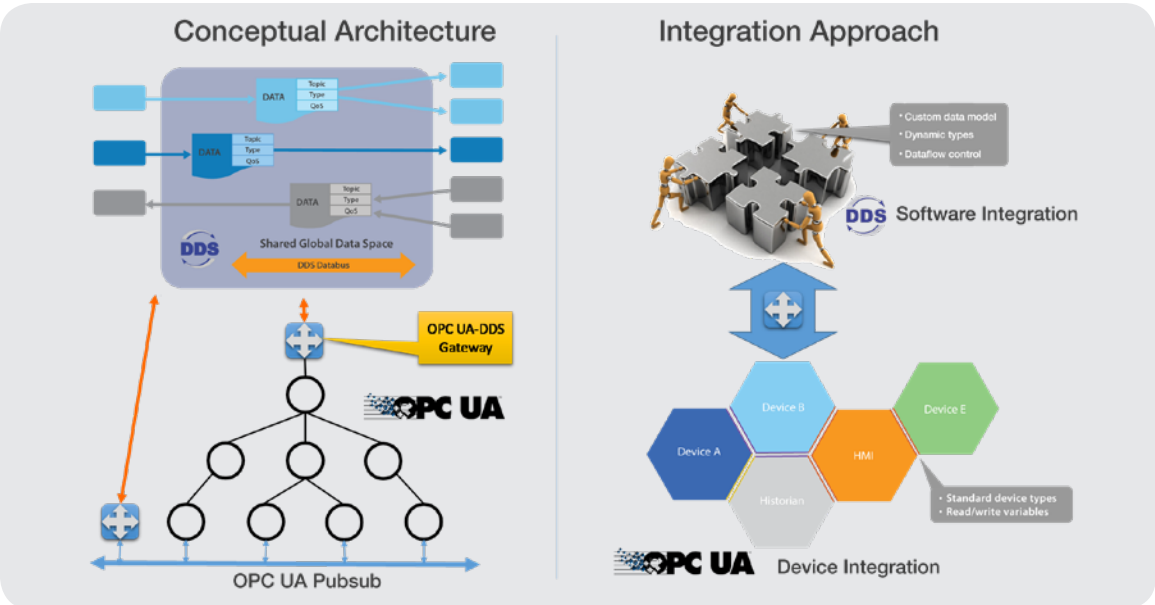


Figure 14. Combined Approach. The Core Connectivity Standard design allows system designers to combine both device and software integration advantages. The standard gateway provides controlled interaction between the systems. This design enables sophisticated software interaction with practical plant operation. Potential applications include autonomous workcells for discrete manufacturing and AI-driven processes in process control plants.

WHY A DATABUS IS SO UNIQUE



A databus is the new kid on the block in the IIoT. Most newbies try to categorize it with more familiar technology, such as publish-subscribe, or a SOA, or a messaging system. Those characterizations, unfortunately, usually detract from understanding. It truly is a new concept. This section attempts to clarify why.

The Change in Mindset to Data Centricity

We'll start with data centricity. As indicated above, data centricity is an architecture formed from participants that communicate only with the data infrastructure. DDS defines a data-centric connectivity framework, or databus. But, it is hardly the only data-centric technology.

The data-centric manifesto, for instance, originated from the critical role of data in the insurance industry. Its principles are instructive here:

- Data is a key asset of any organization.
- Data is self-describing and does not rely on an application for interpretation and meaning.
- Data is expressed in open, non-proprietary formats.
- Access to and security of the data is a responsibility of the data layer, and not managed by applications.
- Applications are allowed to visit the data, perform their magic and express the results of their process back into the data layer for all to share.

It's really a simple and elegant concept. However, data centricity brings power, but also confusion.

Once you catch the data-centric fever, many things suddenly make sense. It's like when you first understand what a database can do. Who would ever design an employee-tracking system, or an auto-parts store, or a supply-chain management system without a database? Yet all these things existed well before the database abstraction came

along. But, it does force a new way of thinking—a data-centric way. Instead of writing applications that create and manage data, you rely on an infrastructure that can store, search and organize data. Before a database, applications would read and write files with special formats, and with lots of special coding to ensure consistency. With a database, applications interact only through the data infrastructure. The database provides a scalable, incorruptible source of truth.



Similarly, for distributed systems, data centricity requires a new way of thinking. Most programming systems and languages are object-oriented (OO). OO flow is fundamentally sequential. You call a method, it returns and then you call the next. This is great for single-threaded programs linked together on one CPU. Since everyone is familiar with OO programming, early distributed systems (like DCOM, OLE and CORBA) simply extended the concept to remote objects. However, even on a single CPU with multiple cores, it is famously difficult to call many methods in parallel.

In a distributed system, this is debilitating. It favors a familiar abstraction at the huge cost of sacrificing parallel performance at scale. It also concentrates state in the objects. That makes it localized: you have to find the object that has the data you want. The real world operates in parallel, so many real-world systems are naturally parallel and data centric. But it takes a real change of mindset to rethink your programming architecture to match. Thus, we force OO abstractions unnaturally onto data-centric systems.

How Does a Databus Work?

The DDS databus implements a connectionless data model that communicates data objects identified by named "Topics." The databus automatically discovers and connects publishing and subscribing applications. Those applications request or offer to interchange Topics with a desired quality of service (QoS). The databus matches Topics, decides if it can meet the QoS, and then effects the communication.

The core architecture supports fully redundant sources, sinks and networks to ensure highly-reliable operation.

It needs no communication servers for discovery or configuration; instead, it connects data sources and sinks through a background "meta traffic" system that supports massive scale. Serverless operation greatly eases configuration while eliminating failure and choke points.

DDS conceptually has a local store of data, which looks to the application like a database table. Each Topic is a row of that table. When you write, it goes into your local store, and then the messages update the appropriate stores on remote nodes. When you read, you just read locally. The local stores together give the applications the illusion of a global data store. Importantly, this is only an illusion. There is no global place where all the data lives; that would be a database. In a databus, each application stores locally only what it needs and only for as long as it needs it. Thus, DDS deals with data in motion; the global data store is a virtual concept that in reality is only a collection of transient local stores.

The QoS matching is unique to DDS. Information producers "offer" to meet QoS commitments; consumers "request" QoS values. The matching simply determines if the offer can meet the request. Once matched, the middleware enforces the "contract." For instance, if a subscriber needs updates 1000 times per second, the middleware will ensure a fast producer is available. The standard defines more than 20 of these QoS policies.

The QoS settings cover most all aspects of communication. They include:

- Selection of best-effort versus reliable delivery modes.
- Desired timing characteristics (latency, deadlines, update frequencies).
- Network usage characteristics (e.g., bandwidth utilization).
- Data Ordering including whether data should be delivered in specific order (e.g., First In, First Out (FIFO) or priority based).
- Data Coherence, the ability to deliver sets of data updates together as a group.
- Lifespan, the ability to expire data when it's "too old."

- History, the ability to control the local data-update cache for each reader and writer, on a per-object basis.

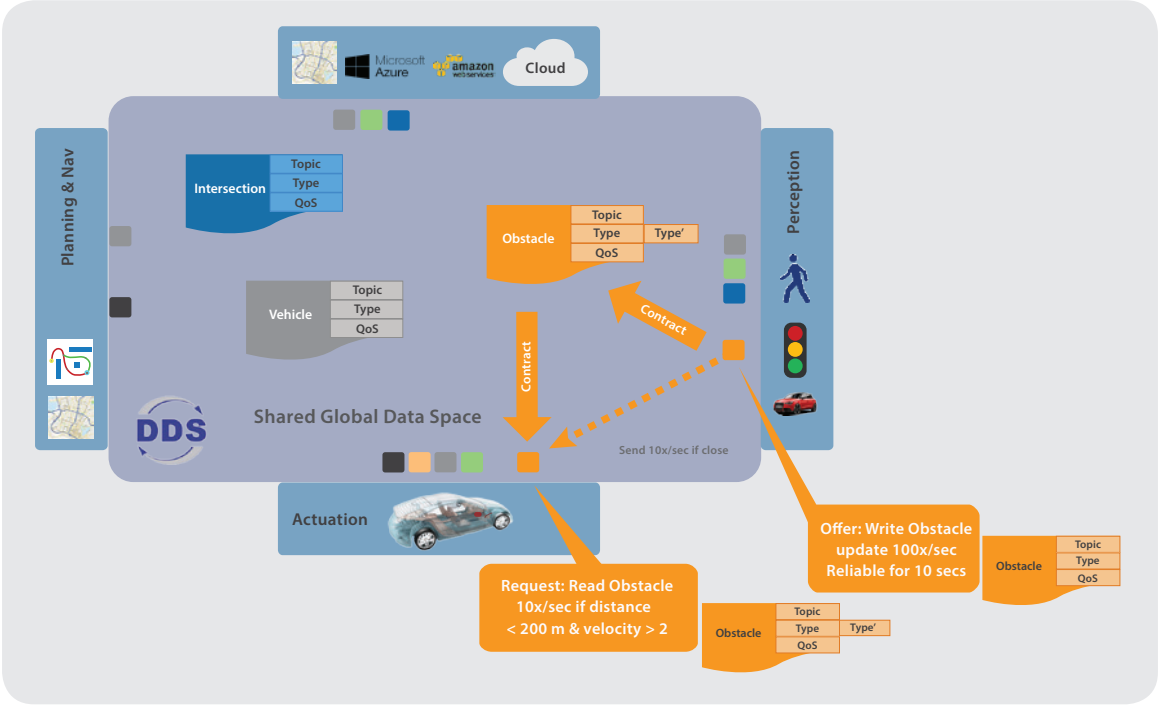


Figure 15. Data-Centric Communications. The databus links any language, device or transport. It automatically discovers information sources, understands data types and communicates them to interested participants. The databus scales across millions of data paths, enforces sub-millisecond timing, ensures reliability, supports redundancy and selectively filters information. Each path can be unicast, multicast, open data or fully secure. In the figure, the traffic monitor will send only "obstacles" that are close to the requester and moving at a rate it can handle.

Figure 15 contains a notional example. In the figure, many participants cooperate in a traffic control system. One of the participants offers to write a Topic called "Obstacle" 100 times a second. It has 1000 buffers, so it can promise to repair lost deliveries for 10 seconds.

The offer itself causes the topic "Obstacle" to appear in the virtual data space; every participant can now access it. Of course, with no requests, no information will be sent; the databus will simply cache the offerer's updates locally for 10 seconds. Then, a reader requests Obstacle updates, but with a filter: it only wants 10 updates a second, and only if they are within 200m and moving faster than 2 m/sec.

Note the type management. Each participant interested in Obstacles provides what it thinks the Type is. Because the databus knows the Type of Obstacle, it can look inside

the data structure to evaluate this filter. In this case, the sender and receiver think the Obstacle Type is different. The databus will analyze the difference and decide if these types are assignable. Assigning incompatible types is defined in the DDS standard; acceptable changes may include things like adding fields, removing fields or inserting dummy fields. The ability to match different versions and types is important to support large distributed systems that may have incremental changes in some devices over time.

The databus analyzes the contracts and decides if the writer can satisfy the reader. In this case, the Types are close enough to be assignable, possibly with modification. The writer is faster than the reader, so that will work. The reader doesn't need reliability, so the writer's cache is irrelevant. The request matches. So, the databus evaluates the filter as data is written, and then sends updates to the requester, at most 10 times a second, only for those Obstacles that meet the filter. The filtering happens at the writer's end for efficiency.

This may sound complicated. But, it is fully automated. RTI has operational systems with 10 million data contracts. There are no servers in this example. Data is sent peer-to-peer, and therefore is fast.

The Database Comparison

The easiest way for most to understand a databus is by comparison with its storage cousin, the database. Both are relational, data-centric technologies.

The new IIC Industrial Internet Reference Architecture v1.8 (IIRA) defines a databus as:

A databus is a data-centric information-sharing technology that implements a virtual, global data space. Software applications read and update entries in a global data space. Updates are shared between applications via a publish-subscribe communications mechanism.

Key characteristics of a databus are:

a) the participants/applications directly interface with the data,

b) the infrastructure understands, and can therefore selectively filter the data, and

c) the infrastructure imposes rules and guarantees of Quality of Service (QoS) parameters, such as rate, reliability and security of data flow.

Like a database, a databus is a data-science concept. Let's look at what it really means. Here are answers to typical questions from those familiar with databases.

HOW IS A DATABUS DIFFERENT FROM A DATABASE?

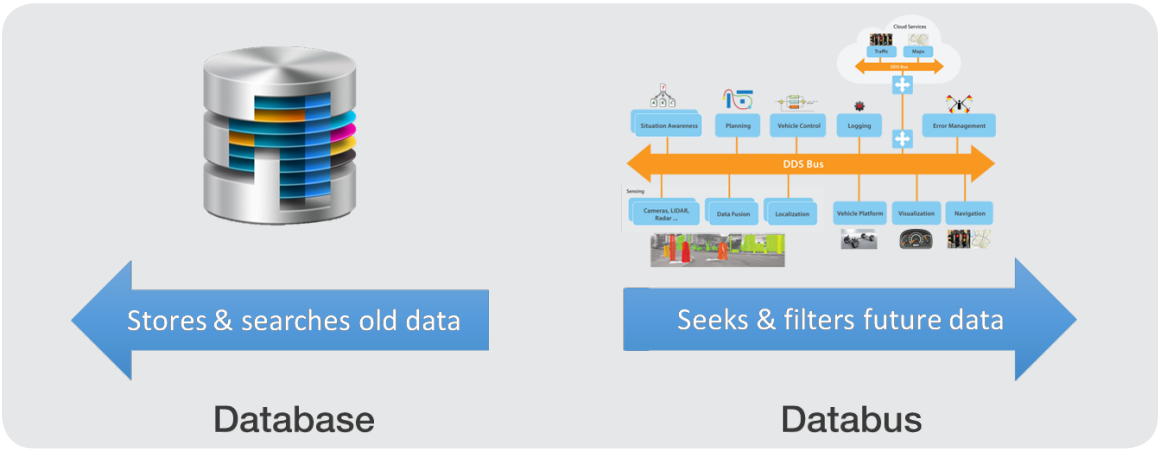


Figure 16. Database versus Databus. A database replaces files with data-centric storage that finds the right old data through search. A databus replaces messages with data-centric connectivity that finds the right future data through filtering. Both technologies make system integration much easier, supporting much larger scale, better reliability, and application interoperability.

The key difference: A database implements data-centric storage. It saves old information that you can later search by relating properties of the stored data. A databus implements data-centric interaction. It manages future information by letting you filter by properties of the incoming data.

Under the hood, a database is a special file. But, user applications don't see that; applications interact with the data itself. A database knows how to interpret the data, enables operations like search and enforces access control. There is no application-level awareness of the file system. Programs using a database read and write and search for data objects, not files.

Similarly, under the hood, a databus sends messages (optimized packets). But, user applications don't see that; applications interact directly with data and "Quality of Service" (QoS) properties like age and rate. The databus controls access to the data, how data in the system changes, and when participants get updates. There is no application-level awareness or concept of "message." Programs using a databus read and write and get updates to data, they do not send and receive messages.

The common property of "data centricity" is hugely powerful in both cases. By enforcing simple rules that control the data model, databases ensure consistency. By exposing the data to search and retrieval by all users, databases greatly ease system integration. By allowing discovery of data and schema, databases also enable generic tools for monitoring, measuring and mining information.



Similarly, a databus understands the content of, and need for, the transmitted data. With knowledge of the structure and demands on data, the databus can do things like filter information, selecting when or if to do updates. The infrastructure itself can control QoS like update rate, reliability and guaranteed notification of peer liveliness. The infrastructure can discover data flows and offer those to applications and generic tools alike. This accessible data greatly eases system integration.

Both the database and the databus implement a relational data model. Thus, applications find information by name or properties of the content. You can search for employees by name or salary in an HR database. You can filter for sensor updates by name or value with a databus.

Databases enabled huge enterprise systems with thousands of applications sharing storage; databases power the enterprise. Similarly, databuses enable huge industrial systems with thousands of interacting applications; databuses will power the IIoT.

IS A DATABUS A DATABASE THAT YOU INTERACT WITH VIA A PUB-SUB INTERFACE?

No, there is no database. A database implies storage: the data physically resides somewhere. A databus is not fundamentally a storage technology.

The databus defines how to interact with future information. For instance, if "you" are an intersection controller, you can subscribe to updates of vehicles within 200m of your position. Those updates will then be delivered to you, should a vehicle ever approach. Delivery is guaranteed in many ways (start within .01 secs, updated 100x/sec, reliable, etc.). The data may never be stored at all; it's just delivered.

ISN'T A DATABUS THE SAME AS PUB-SUB?

No, pub-sub is a pattern while a databus is an architecture that may support many patterns. Most communications on a databus are pub-sub interactions. But the key isn't the pattern, it's the control of that pattern. In fact, the DDS standard codifies both publish-subscribe and request-reply patterns. Some vendors, including RTI, also support queuing. Pub-sub is great for parallel data access. Request-reply best fits one-to-one interactions like commands. And queuing is great for load balancing, for instance in large websites. The databus can work with all of them. The difference is that the interaction is with the data, not an active entity. So, for instance, when using request-reply, you are requesting the data to find a reply from any source, not asking for a response from a particular server.

Most pub-sub is very primitive. An application "registers interest," and then everything is simply sent to that application. So, for instance, an intersection collision-detection algorithm could subscribe to "vehicle positions." Pub-sub sends messages from any sensor capable of producing positions, with no knowledge of the data inside that message. Even "content filtering" pub-sub offers only very simple specs, and requires the system to pre-select what's important for all. There's no real control of flow.

A databus is much more expressive. That intersection could say "I am interested only in vehicle positions within 200m, moving at least 2m/s towards me. If a vehicle falls into my specs, I need to be updated 200 times a second. You (the databus) need to guarantee me that all sensors feeding this algorithm promise to deliver data that fast–no slower or faster. If a sensor updates 1000 times a second, then only send me every 5th update. I also need to know that you actually are in touch with currently-live sensors (which I define as producing in the last 0.01secs) on all possible roadway approaches at all times." (These are a few of the 20+ QoS settings in the DDS standard.)

Note that a subscribing application in the primitive pub-sub case is very dependent on the actual properties of its producers. It has to somehow trust that they are alive (!), that they have enough buffers to save the information it may need, that they won't flood it with information nor provide it too slowly. If there are 10,000 cars being sensed 1000x/sec, but only 3 within 200 meters, it will have to receive 10,000*1000 = 10 million samples every second

just to find the 3*200 = 600 it needs to pay attention to. It will have to ping every single sensor 100x/second just to ensure it is active. If there are redundant sensors on different paths, it has to ping them all independently and somehow make sure all paths are covered. It also has to know the schema of the producers, etc.

The application using a databus will, by contrast, receive exactly the 600 samples it cares about, comfortable in the knowledge that at least one sensor for each path is active. The flow rate is guaranteed. Sufficient reliability is guaranteed. The total dataflow is reduced by 99.994% (we only need 600/10m samples, and smart middleware does filtering at the source). The databus delivers only the needed information.

"THE PARTICIPANTS/APPLICATIONS DIRECTLY INTERFACE WITH THE DATA." COULD YOU ELABORATE ON WHAT THAT MEANS?

With "message-centric" or "object-oriented" middleware, you write an application that sends data, wrapped in messages, to another application. You may do that by having clients send data to servers, for instance. Both ends need to know something about the other end, usually including things like the schema, but also likely assumed properties of the data like "it's less than .01 seconds old," or "it will come 100x/second," or at least that there is another end alive, e.g., the server is running. All these assumed properties are completely hidden in the application code, making reuse, system integration and interoperability really hard.

With a databus, you don't need to know anything about the source applications. You make clear your data needs, and then the databus delivers it. Thus, with a databus, each application interacts only with the data space. As an application, you simply write to the data space or read from the data space with a CRUD (Create, Read, Update, Delete) interface. Of course, you may require some QoS from that data space, e.g., you need your data updated 100x per second. The data space itself (the databus) will guarantee you get that data (or flag an error). You don't need to know if there is only one or 27 redundant sources of that data, or if it comes over a network or shared memory, or if it's a C program on Linux or a C# program

on Windows. All interactions are with your own view of the data space. It also makes sense, for instance, to write data to a space with no recipients. In this case, the databus may do absolutely nothing, or it may cache information for later delivery, depending on your QoS settings.

Note that both database and databus technologies replace the application-application interaction with application-data-application interaction. This abstraction is absolutely critical. It decouples applications and greatly eases scaling, interoperability and system integration. The difference is really one of old data stored in a (likely centralized) database versus future data sent directly to the applications from a distributed data space. Both technologies implement fundamental decoupling.

HOW EXACTLY IS A DATABASE LIKE A DATABUS?

As data-centric technologies, the database and databus have many similarities. Both have the same fundamental benefits, data access and lifecycle control. Both also have integrated security; applications do not have to do anything special to be secured.

Aspect	Database	Databus
Purpose	Stores old information	Seeks future information
Underlying OS implementation	File	Messaging system
Data organization	File format	Protocol
Implementation paradigm	Centralized	Distributed
Base interaction pattern	Query/response	Publish/Subscribe
Guarantees	ACID (data integrity)	QoS (data availability & flow)
Data structure	Schema (data table)	Schema and timing (Data model)
Atomic data unit & index	Record, indexed by keys	Instance, indexed by keys
Fundamental benefit	Scale, reliability, truth	Scale, reliability, truth
Data lifecycle	CRUD	CRUD
Data access	Relational (structured query)	Relational (distributed filter)
Security	Part of architecture	Part of architecture

Table 3. Databus and Database Concept Map. The database is the storage cousin of the databus. Both are data-centric, relational technologies. Applications interact only with the data through a controlled interface with integrity guarantees.

The table compares concepts between the two. Key similarities include the concept of data lifecycle and relational data access. The major difference is centralized storage versus distributed interaction.

The guarantees are important differences. It is critical that both need to supply the system with a clearly-defined source of "truth." In the database case, the truth is about a consistent view of stored data, known in the industry as "ACID" properties. For a databus, truth is more about guaranteed availability and flow rates of data. In both, truth in a large system enables scale and reliability.

The Layered Databus Pattern

THE VALUE AND CHALLENGE OF HIERARCHY

Large organizations of people, such as companies and governments, always form hierarchies. Dividing a system into subsystems isolates changes between components, reduces coupling and eases team coordination. Large systems require a "divide and conquer" hierarchical approach.

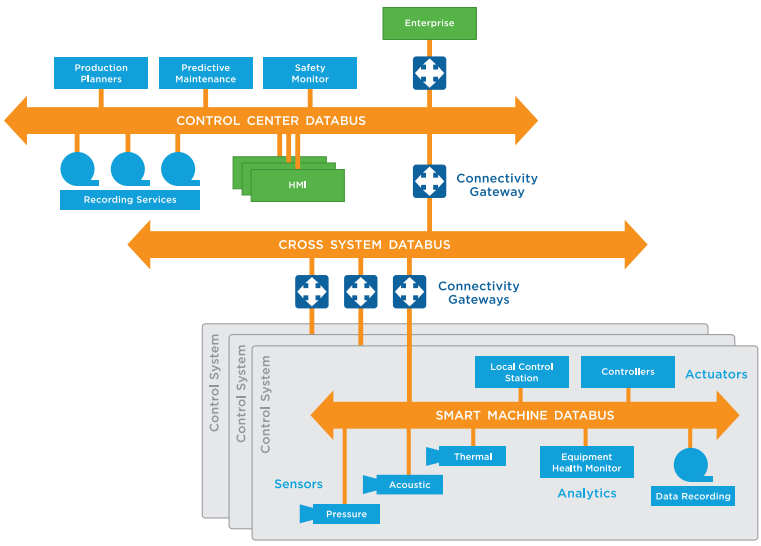


Figure 17. Layered Databus Pattern.

Unfortunately, most layering technologies also introduce barriers to information flow. The worst barriers require static configuration, introduce subtle dependencies and cause inefficient flow patterns. In short, like human organizations, computer architecture hierarchy enables scale but with risk of inefficiency.

Of course, in modern organizations, the database is the great enabler of efficiency. Providing clear data access to all parts of the organization increases efficiency dramatically. Can a databus do the same for IIoT systems?

The value is compelling. The databus –"all data everywhere"– abstraction is elegant and efficient. It is critical in a subsystem. And easy, efficient flow is even more valuable as it extends to larger and larger systems. Fortunately, the global data abstraction can extend beyond a local domain through intelligent "gateways." The IIC's seminal works on architecture and connectivity call this the "layered databus" architectural pattern. This pattern supports a wide variety of system types.

ROUTING THROUGH GATEWAYS

Hierarchy is valuable. Of course, hierarchy also poses challenges including enabling discovery, managing security and controlling dataflow. The trick is deciding what information to pass through each layer of the hierarchy.

To maintain the global data space abstraction, we cannot just implement a patch between layers. That would force applications to talk directly with the gateways instead of to the data. Instead, we need something that acts as a completely transparent dataflow controller between levels. It must find and deliver data across levels as if that data was fundamentally available (in the limited needed form) at each level.

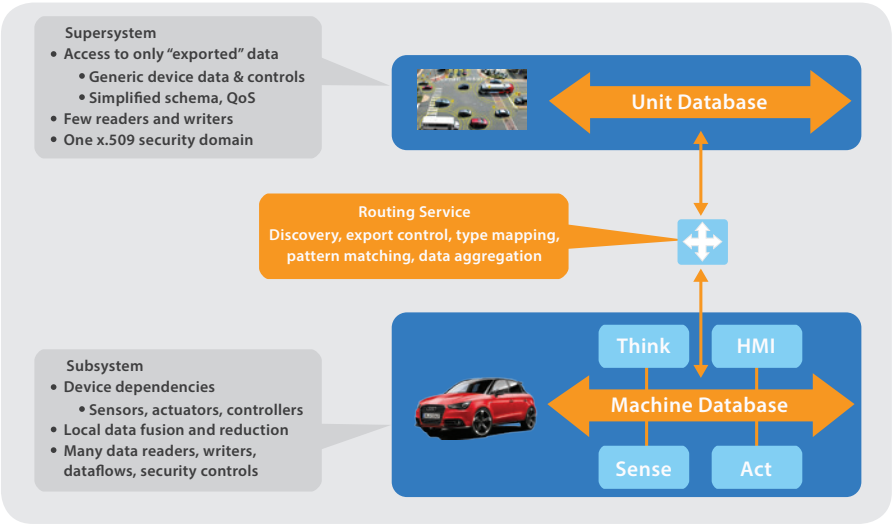


Figure 18. Routing Service Gateway. The Routing Service maps between databus domains. To the supersystem, each subsystem appears as a single participant and security domain. The subsystem has control over data export and access properties. This pattern preserves the global dataspace abstraction for the exported data.

There is an analogous concept in enterprise networking. Most enterprise networks, such as web services, are Service Oriented Architectures (SOAs). SOAs provide well-defined services to applications. They must maintain this abstraction across hierarchical levels.

To exist at scale, SOAs need the concept of an Enterprise Service Bus (ESB). An ESB acts as a switch between interacting applications, matching service models like client-server to pub-sub or database access. These switch paths are called "routes," and the ESB is said to be "routing" service requests. For clarity, this type of "routing"

is not related to packet routers like those sold by Cisco and Juniper. The ESB routes service accesses between applications that may not otherwise work together.

SOAs are not data-centric, they are service-centric. Just as an SOA must route service requests between levels, a data-centric system must route dataflows between levels.

RTI implements this with a special service called the Routing Service (RS). Note that RS is a service, rather than a server. The difference is that multiple RSs can be easily implemented redundantly. Each RS is simply a databus participant that lives on two databuses. The possibly-redundant dataflow is resolved on each side in the usual way.

In both SOAs and data-centric systems, routing is critical to a scalable, decoupled system architecture. It fundamentally preserves the abstraction across systems-of-systems, allowing them to cleanly interact.

CLOUD ACCESS

The Layered Databus architectural pattern extends the "all data everywhere" to the cloud. Using DDS to access the cloud has many advantages over other approaches.

For instance, DDS supports many transports. If your application can use UDP, it can scale exceptionally well, take advantage of QoS settings, and deal intelligently with intermittent sending and data throttling. But DDS runs fine over other transports, including TCPv4 and TCPv6. Most applications use the TCP transports to simplify firewall traversal. Complex systems need many transports; DDS routes between them. For reliability, it can also automatically resolve multiple, redundant transports. All these things are invisible to applications that (of course) only see data relationships. Any transport that can deliver the requested data QoS works.

The usual cloud technologies are easily integrated. There is an official OMG standard and many products for connecting to RESTful web services. Vendors offer other options, including simplified and powerful access to scripting languages like node.js, JavaScript, Python and Lua.

More recently, vendors are offering advanced functionality useful for cloud applications. For instance, RTI supports

"Transport Mobility" that preserves reliability guarantees across IP address changes for moving equipment, and, especially when paired with load balancer hardware, can support hyper-busy web sites. "Topic Query" allows cloud application access to historical data. These can work together; see the auonomous vehicle example application below.

Security is a particular challenge between edge and cloud. Many systems rely on IPSec "VPN" networks that protect from outside intrusion at the cost of opening all connections to potential inside attack. The "layer 2/3" security is better, because it locks communications to just one pipe. DDS supports both of these approaches.

However, DDS also offers better options. DDS per-data-stream fine-grain security can extend all the way to the cloud. This protects system dataflow, whether the applications run on a tiny embedded processor, a powerful edge computer or an elastic cloud service. Most importantly, it preserves the same simple abstraction across device, fog and cloud.

LAYERED DATABUS EXAMPLES

The layered databus pattern permits scale with explicit dataflow control. It is useful in many contexts. The figures below outline examples in large SCADA (Figure 19, NASA's launch control system), large hospital networks (Figure 20, GE Healthcare's clinical databus design) and in the massive Grand Coulee Dam power plant (Figure 21).

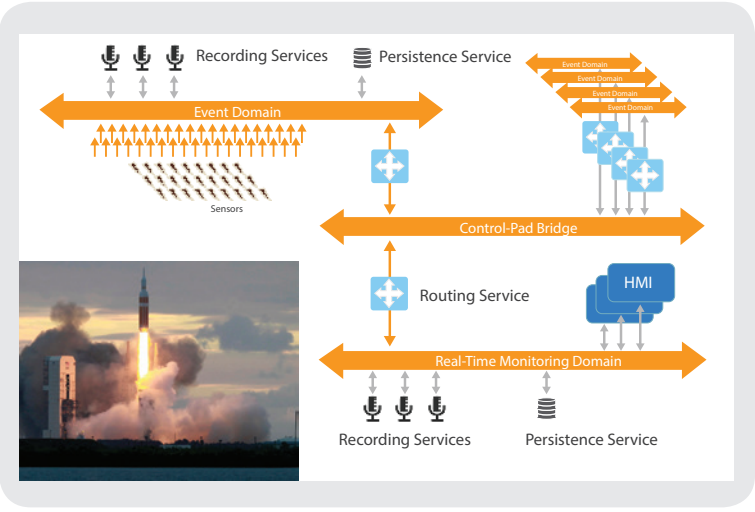


Figure 19. Launch Control Architecture. NASA's Launch Control System for Orion is the world's largest single-system SCADA, with over 300,000 points. It essentially operates as a special-purpose chemical plant. The rocket is placed on the pad unloaded; the system then pumps explosive fluids to charge the thrusters, tests the various electrical and safety systems and prepares the complex payload. All sensor data is viewable in real-time (after downsampling) on the HMIs in the control room.

The databus global dataspace abstraction simplifies the architecture. Local control systems may require fast loops and sensor redundancy resolution. The control room needs on-demand access to any of many thousands of sensors and actuators on the launch pads. In addition to the Routing Services that bridge domains, the system uses Recording Services to capture data for forensic use and Persistence Services to ensure data durability.

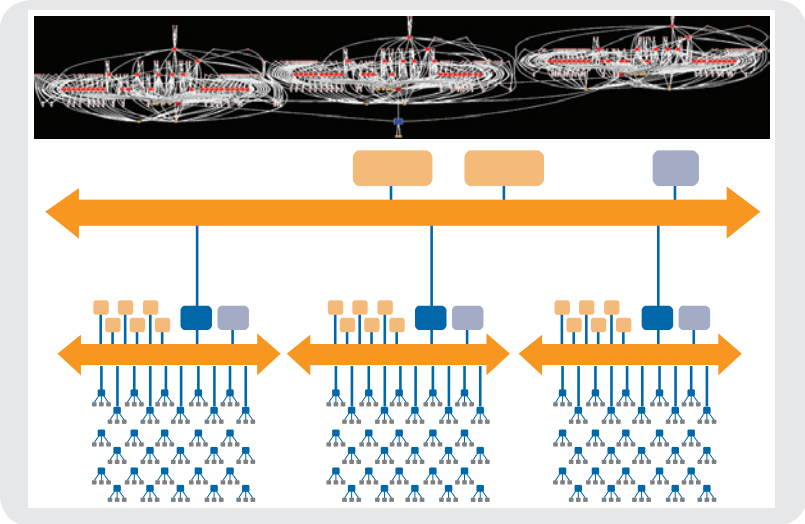


Figure 20. Hospital Databus Architecture. Hospitals are complex IT environments. Soon, each patient will be connected to a set of devices working together, and with an AI. This 24x7 connected system will help human care teams provide much better care than today's singleton disconnected devices can.

But that causes a huge scaling problem. In this figure, there are less than 100 rooms. Each could have 10 or more devices. A large hospital has more than 1000 rooms and 200,000 devices.

Each floor, ward and room could be controlled by a databus. The diagram represents the layered databus architecture for the system in Figure 20. In the test results at the top of the figure, each of the 300 red dots is a Routing Service, each controlling a room full of equipment.

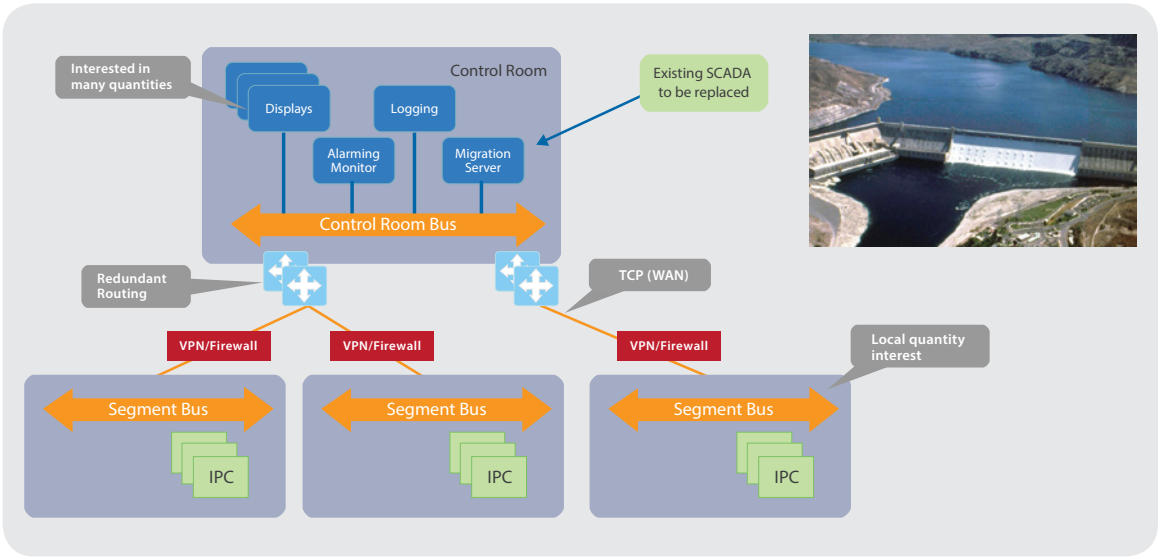


Figure 21. Hydropower Plant Architecture. With 6.8 GW of capacity, the Grand Coulee Dam is the largest power plant of any type in North America. It uses a hierarchical databus pattern to ensure reliable operation, even in the face of multiple failures. Redundant Routing Services are a key part of the design; they ensure data connectivity. The dam, with its control infrastructure and associated switchyards cover a large geographical area. Because the databus design isolates participants from network and processor differences, it can seamlessly combine many different subsystems, both local high-performance networks and wide-area TCP connections. To participants, all data appears simply as local CRUD-accessible memory objects.

THE IIOT SECURITY CHALLENGE

The IIoT is a brave new world for risk management. There are all sorts of new opportunities for attack. The hack that allowed remote control of a Jeep over the Internet is a classic example. More concerning industrial cases include the Stuxnet worm that destroyed Iran's nuclear program, the European grid backdoor installed by malware, and the Excel spreadsheet exploit that caused a blackout in Ukraine.

That said, intelligent machines also have more opportunity to protect themselves. The sad truth today is that most systems are poorly protected (like that Jeep). Security gets orders of magnitude more attention today than only a short time ago. Most industrial systems didn't even consider anything beyond "eggshell" firewalls or "air gap" offline designs until recently. That has changed 100% today; everyone is thinking security, security, security. And the progress is exhilarating.

That said, in my experience (and this may shock security wonks), security is not a change driver. Fear is simply not enough. Industrial systems are usually not willing to implement a new architecture (just) to improve security. The power industry is my favorite example. The industry has been screaming for 20 years that security is a problem. And, they will go right on screaming...unless something else drives the change.

The good news: the IIoT is that change driver. And while security may not drive the change, security is absolutely a change gate. When implementing a new architecture for any reason, every application insists on security. Since the IIoT is motivating many, many industrial applications to redo their architectures, security is getting better. Of course, implementing a new architecture for a major industrial application, or for that matter an entire industry, is daunting. But this is the magic of the sweeping changes offered by the IIoT. The IIoT is compelling. Change is coming, and it's coming fast.

Is risk increasing? The real question isn't if connecting a system exposes it to more attackers; that's obvious. The real question is if installing an IIoT design makes your

system safer or riskier. Because of this new driver and awareness, we are probably decreasing our collective risk profile over time. This is simply because the process of implementing an IIoT solution motivates a badly-needed security audit. The greater attention means we are installing cyber "burglar alarms" faster than the rise in "burglars." So, our collective "likely real risk" of a major industrial infrastructure event is probably decreasing.

Many consider this an overly optimistic view of the future. I counter that they hold a too-optimistic view of the present. You see, the situation today is unacceptably, intolerably, unbelievably high risk. Entire industries run without an ounce of security. It seems scarier in the future only because the risk you don't know seems worse than the risk you do know. That's human nature. But, anyone who looks will see that the current risks are very high, and the new designs are much better.

How much better? There are sweeping improvements in both understanding and technology that enable better security.

The Industrial Internet Consortium (IIC) recently published the most complete treatment of security in the industry, the "Industrial Internet Security Framework" (IISF). This 162-page tome outlines the security challenge from the broadest concept of trustworthiness to the key considerations in implementation. Perhaps its greatest contribution is simply the cataloging of all the deep and wide considerations of the problem. Even a casual perusal will increase designer's awareness of the challenges.

Security and Data Centricity

More specifically, the most exciting change is in the development of data-centric security.

Data centricity is compelling. As we've been discussing, many potential IIoT systems primarily face scalability and system integration challenges. With a little thought, the architects figure out that IIoT systems are all about the data, and then that they really have a high-performance data flow and data transparency challenge. The best way to provide transparent flow is with a data-centric databus. It scales well, performs well, and connects low-level control to high-level intelligence.

Unfortunately, this realization leads to a "tail-wagging-the-dog" dilemma. The architecture is the "dog," systems need the simplicity and performance of a communications pattern that directly sends the data where it's needed, right now. That data transparency makes large-scale future IIoT systems manageable.

So, the "dog" side of the dialog goes something like this:

> Hey! Let's just send the data right where we need it. Pervasive data availability makes systems fast, reliable and scalable. And look how much simpler the code is!

Of course, although data transparency is an integration dream, it's a security nightmare. So, here comes the security "tail":

> We can't maintain thousands of independent secure sessions! How do we keep such a system secure?

To answer this question, the DDS standard developed a security architecture that exactly matches its data-centric dataflow design. The DDS security specification is the newest of the major connectivity security specifications. It includes the latest in security technology, including authentication, access control, confidentiality, non-repudiation, tagging and logging. Its "plug-in" design allows it to adapt to different applications, from simple "check-the-box" security to extremely custom environments. But its most important innovation is mating security to data centricity.

The result? The data-centric dog wags its perfectly-matched data-centric security tail. Security works seamlessly without clouding data transparency.

Security that matches the architecture is elegant and functional. Even better, because it's a standard, products are coming to market that enable secure interoperability between vendors.

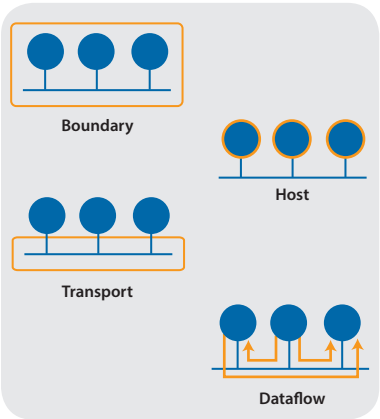


Figure 22. Defense in Depth. The best security uses layers that progressively make it harder to attack the system. Most distributed systems implement some form of the first three layers. Only data-centric systems can control dataflow and implement the last.

Most systems implement boundary security with firewalls and airgaps. Unfortunately, these are easily breached by insider attacks or Stuxnet-like tricks. There are several protective host security measures like secure passwords, white listing and port locking. Most distributed systems implement layer 2 or layer 3 transport encryption. However, transport security is both bidirectional and open between any applications at the endpoints. Only dataflow security can selectively allow reading and writing of particular dataflows.

Dataflow security fundamentally adds another layer of protection. It is hard to attack without a detailed understanding of the application data structure.

How DDS Handles Security

DDS handles security similarly to the way it handles QoS. To enable, security is affected simply by turning it on and providing three certificates to each device:

- Each participant needs an “ID” certificate so it can prove it is valid.
- Each participant needs a copy of the “governance” certificate that configures security itself, including whether or not headers are to be encrypted, etc.
- Finally, each participant needs a “permissions” certificate that says exactly which topics it can produce or consume.

In most systems, the last two certificates can be the same for all participants.

DDS security perfectly matches the databus architecture. In fact, the security design has no API, so there is no code to write. Even more interesting, each topic flow is individually controllable, including direction of data flow. The security simply adds another condition to the dataflow control provided by the other QoS settings.

A DEEPER EXAMPLE: AUTONOMOUS VEHICLES (CARBOTS)

Why Do We Need Carbots?

Tons of metal do not collide quietly.

I started my career crashing cars; my first job was in biomechanical impact testing at a research laboratory at the University of Michigan. Our goal was to develop occupant protection: better airbags, child car seats, side impact systems, even wheelchair restraints.

What I best remember from our testing is the "BANG!" at the end of the track. That shocking noise always resulted in twisted metal and obvious (simulated) carnage.

When I worked there, 45,000 people were dying on the roads in the US each year. Last year, 40,000 died. We have improved, but only incrementally. That's despite multi-stage airbags, crash-absorbing structures, anti-lock brakes, electronic stability control, child seats, belt pretensioners, seat-belt usage programs, better roads, DUI penalties, drunk-driving ad campaigns, automated signal cameras, and on and on. We fixed many safety weak spots. But, we didn't fix the #1 safety system weakness: the driver.

Now we can. Today, we have the opportunity to automate driving. No...we are faced with the moral imperative to automate driving. Autonomous cars can decrease injuries and fatalities. There are few more noble pursuits in technology.

Some worry about the moral choices that a carbot must make. The dilemma typically flows along these lines: "Imagine the carbot is driving and a stray truck crosses its path. It can choose to kill you (the occupant) or hit five pedestrians." What should it do?

My response: what would you do as a driver? Oops! BANG! You are dead, or they are dead, or both. You may have even swerved into a crowd of kindergartners in your panic. You see, you need to make that choice in about half a second. Your moral consternation is unlikely to yield an optimal result no matter what you think you should do. People simply cannot react fast enough to execute that decision. If we must examine such a situation, let's be encouraged by the fact that (someday) a smart carbot may actually have the processing speed to make any choice. You, as a human driver, do not.

But, this also misses the key point. The moral imperative is not to decide if a carbot should choose to kill you or five pedestrians in some absurdly unlikely situation. The moral imperative is to stop the certain death of tens of thousands of people, this year, next year and every year. People text, drink, argue, fall asleep, talk on the phone, run lights, drift across lanes and drive through stop signs with alarming regularity. Even early carbots will easily be better than human drivers. We need to stop debating right and wrong in rare situations and do what's clearly right in common situations – in fact, so common that they don't even make the news. If we choose to embrace them, carbots will save thousands every year. That's our moral choice. That's our only choice.

The Challenge of Autonomy

Intelligent vehicles are complex distributed systems. An autonomous car combines vision, radar, lidar, proximity sensors, GPS, mapping, navigation, planning and control. These components must combine into a reliable, safe, secure system that can analyze complex environments in real time and react to negotiate chaotic environments. Autonomy is thus a supreme technical challenge. An autonomous car is more a robot on wheels than it is a car. Automotive vendors suddenly face a very new challenge.

Of course, the challenge isn't completely new; autonomy has a rich technology heritage in other industries, including aviation drones, space robotics and underwater vehicles. Autonomous cars can leverage this base, but streets and highways are much faster, more chaotic environments. Human cargo raises the stakes for reliability and safety. Passenger vehicles are already very complex systems. Thus, cars take the autonomy challenge to a new level.



Figure 23. Data Centricity in Autonomous Systems. Data-centric connectivity originated in autonomous robotic systems. Researchers building autonomous air and space vehicles at the Stanford Aerospace Robotics Laboratory developed the core technology. Early RTI middleware applications include most US unmanned air systems ground stations, dozens of NASA rovers, and Advanced Driver Assistance Systems and autonomy for Volkswagen research. RTI runs autonomous systems today in air, land, sea and space.

They need industrial-class reliability, true real-time response and powerful system integration.

In all of these systems, DDS provides real-time performance, high reliability, system integration, precise dataflow control and complex software decoupling. It greatly accelerates and simplifies distributed system development. DDS is the only technology that can deliver microsecond latency, IEC 26262 safety certification, top security and proven operation in production product lines.

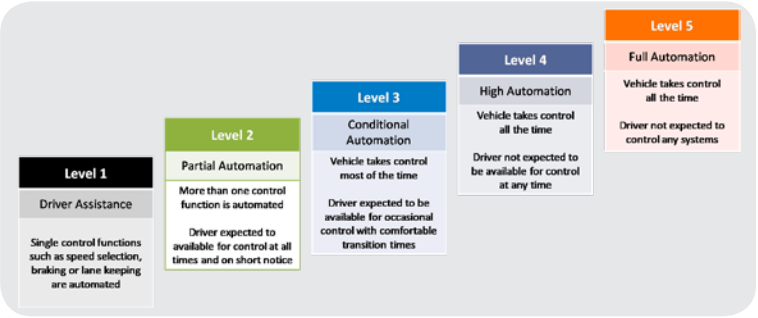


Figure 24. Levels of Autonomy. Autonomy is a sliding scale. The levels differentiate by the amount of time between human intervention.

Autonomy Demands

There is a world of difference between the levels of autonomy shown in Figures 24 and 25. Level 1 systems require almost no intelligence. "Smart" cruise control, for instance, is doable with a simple feedback loop wrapped around a single sensor. Level 2 systems are "hands off, eyes on." The driver must be constantly alert. Level 2 is much harder than Level 1, but still does not require sophisticated sensing, situational awareness, or planning. These are evolutionary designs known as Advanced Driver Assistance System, or ADAS. They assume an alert driver at all times.

Level 3 targets "hands off eyes off"; the driver does not have to pay attention. It works only in well-controlled environments like freeways. Level 3 requires most of the sophistication of full autonomy, but without advanced real-time perception needed to understand a complex city environment.

Level 4 is the current golden target of automotive systems. These will be capable of operating on city streets with complex interactions. That requires advanced perception, data fusion and processing. Level 4 systems are independent but require a driver. They fall back on the human driver in tough situations, ideally rarely. A Level 4 vehicle may, for instance, only need help getting around construction zones.

Level 5 doesn't require a driver in the car at all. The carbot can handle any reasonable situation autonomously.

It is worth pointing out that autonomous air vehicles have been operating at Level 4 for many years. Drones and even passenger-jet flight management systems are capable of flying entire missions or routes with no human intervention. Of course, the environment is much simpler; there's not much to hit at 35,000 feet.

That highlights a key point: environmental complexity is key. Autonomous cars can operate in very chaotic urban environments or on very structured freeways. Structured environments make sensing and situational analysis much easier. Much of the complexity is independent of speed. Parking lots, for instance, are rife with unexpected obstacles, pedestrians and sensing challenges. Autonomy will be practical on structured freeways, even at high speed, before it works reliably in general driving.

Autonomy Architecture

The key to advanced autonomy is the simple realization that system components interact through data flow. Complex systems, therefore, operate by controlling that dataflow exactly as needed by each component. This is, of course, data-centric connectivity.

As of this writing, there are many vehicles approaching Level 3 autonomy. This relatively simple challenge can use a centralized design that sends all data to a single

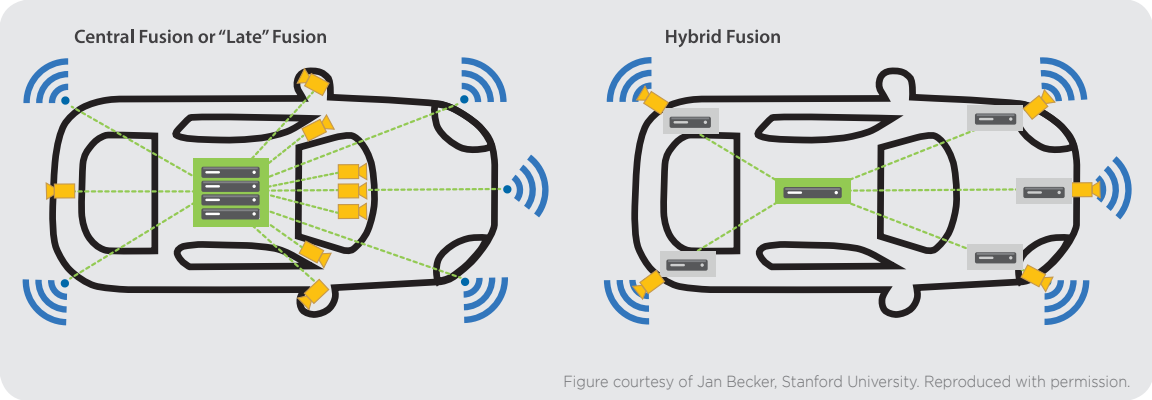


Figure courtesy of Jan Becker, Stanford University. Reproduced with permission.

Figure 25. Sensor Fusion. Perception modules must fuse multiple sensors to better model the world. Level 3 autonomous systems can send all data to a powerful central computer. However, this design breaks down for the complex sensor architectures required for more demanding autonomy at levels 4 and 5. These systems benefit from a distributed design that processes data closer to the sensors, thus spreading the workload, reducing wiring and allowing redundancy.

central computer, sometimes called “late fusion.” Consider, for instance the perception system. A key need is to “fuse” data from many sensors into a single common understanding of the situation. The easy way to access all that data is to get it all to the same place for processing. Thus, if the vehicle has an array of cameras and proximity sensors, each would send the raw video and data streams to the central processor. This simplifies computing. Of course, the processor becomes a choke point and single-point-of-failure. Worse, it requires a lot of data transfers and dedicated video wiring for each camera.

For higher levels of autonomy, this design breaks down. Distributing computing to the edge of the vehicle can greatly decrease dataflow demands. It can also enable redundancy, vastly faster sensor systems and more.

To fuse sensors, the intelligent modules still need the data, of course. However, rather than send all the raw data to one central box, it’s more effective to pre-process the data closer to the sensor. This is often called “early” or “hybrid” fusion. Fundamentally, it leverages the power and redundancy of distributed compute and sensor systems to affect a capable and reliable computing architecture. The databus is a good fit for this design; its data-centric virtual “all-data-everywhere” abstraction is a powerful substitute for actually sending all the data to a central node.

In fact, the databus concept was developed for autonomy. It has unique properties that map well to the challenge. Let’s take a look at some of those properties by examining actual applications. Table 4 summarizes the challenges and approach.

Autonomous System Challenge	Databus Approach
Manage Complex Dataflow	Global Data Abration with QoS Control
Bandwidth Management	Conditional Dataflow
Reliable Data Availability	Easy Redundancy without Servers
Guaranteed Real-Time Response	Peer-to-Peer, Multicast, Fallback, TSN Useable
Complex System Integration	Explicit Interface Control and Data Model Matching
Integrated Security	Dataflow - Level Secure Flow Control
Flexible Deployment	Location, Chip, Language, Network, OS Decoupling
Affordable Safety Certification	Certified Module Separation

Table 4. Autonomy Challenges. Autonomous systems pose unique demands on system architecture. The databus concept directly targets this application, and thus has unique approaches for each demand.

HANDLE COMPLEX DATAFLOW WITH ONE ABSTRACTION

Complex systems must support many dataflows. Most earlier designs require a separate technology for every flow. For instance, older systems would send the extreme throughput of video over a streaming protocol, but use a specialized real-time bus for small but frequent control signals. Worse than the hardware, this also forced a change in abstraction. That made it hard to write software that could fuse the sensor sets.

Autonomous systems make this much more challenging. Carbots combine many sensors, perception modules, intelligence, direct and derived sensors, feedback control, and off-vehicle communications. A carbot with cameras, Lidars, radars, ultrasonic sensors, GPS location and feedback control can easily generate 1.5 GBytes/second or 100 Terabytes per day. Combining all this data through traditional designs is extremely messy.

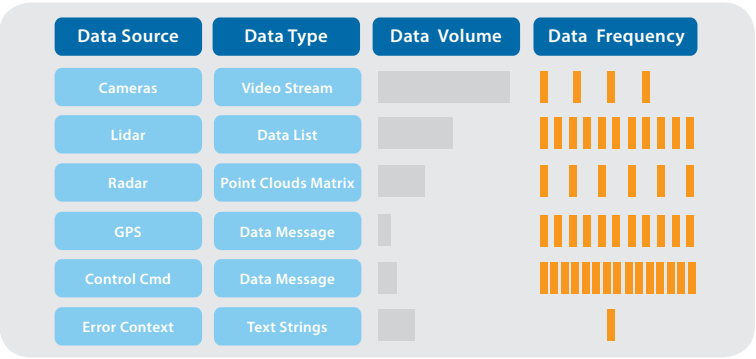


Figure 26. Complex Dataflow. A complex system must support many data types and sources. Some are very high volume; others are fast. QoS control permits each module to specify every data relationship required for operation. The specifications are much more than just what data is needed. They include QoS like update rates, reliability, data availability guarantees and more. Infrastructure that specifically sends exactly the right information to exactly the right places at the right time makes system development much easier.

As shown in Figure 26, QoS control allows a single abstraction to handle all flows. To be fair, QoS configuration can be tricky. But the fundamental capability to handle everything in one framework greatly simplifies all the components. Distributed fusion architectures are much easier with an abstraction that delivers the right data to each module.

BANDWIDTH MANAGEMENT AND CONDITIONAL DATAFLOW

The databus is also smart about sending data only when needed. This allows development teams, for instance, to simply publish debugged data all the time. Of course, a databus will not actually send anything unless a debugging tool is connected.

Recently, this capability has taken yet another step towards full data centricity. The last RTI release offers a feature called “Topic Query.” It lets an application join and retroactively subscribe to historical data in depth.

Topic Query helps autonomous vehicle systems collect the data they need for training sets. Carbots can generate a lot of data; estimates range up to tens or even 100 Terabytes per day. However, 99.9% of that is uninteresting video of the car in front. It’s completely impractical, and silly, to send it all to the cloud. Every now and then, of course, there is a terribly interesting event like a near-miss or even a collision. When that happens, it’s important to get all the data to the cloud so the AI algorithms can learn from it and avoid future similar mistakes.

This is a classic case of conditional dataflow, easily handled with the data-centric global data abstraction in real time. With the addition of Topic Query, this also works after the fact. The carbot can simply store all the data locally. When there’s an important event, a cloud-based system can request all the detailed data just for the event. The databus will send everything as if it was subscribed all along.

ENSURE RELIABLE DATA AVAILABILITY

Autonomous carbots cannot stop, even for a few milliseconds. A vehicle traveling down the highway at 200mph (they will soon) covers nearly a football field every second. That’s no time to reboot a server. A safe vehicle can’t have any single points of failure.

A databus design supports this availability by making it easy to implement a fully-redundant architecture, including computing, sensors, networking, storage and software.



Figure 27. IIoT Reliability-Critical Applications.
Hydropower dams can quickly modulate their massive power output by changing water flow rates, and thus help balance the grid. Even a few milliseconds of unplanned downtime can threaten stability.

Air-traffic control faces a similar need for continuous operation. A short failure in the system endangers hundreds of flights.

A proton-beam radiation therapy system zaps tumors with protons traveling at 70% of the speed of light. That requires a precise control of a superconducting synchro-cyclotron particle accelerator. This complex system must guarantee precise operation without operational dropouts during treatment.

Since it communicates state instead of sending messages, databus systems naturally support multiple producers and consumers of every dataflow.

Advanced implementations can do much more. For instance, RTI software doesn't need any servers. It allows redundant network transports by being intelligent about delivering data only once upon receipt. Thus, adding redundancy is simple. This is important well beyond autonomous vehicles. Figure 28 shows some examples.

GUARANTEE REAL-TIME RESPONSE

There are many ways to characterize "real time." All systems should be "fast." To be useful, we must specifically understand which timing requirements drive success.

"Real time" is much more about guaranteed response than it is about fast. Many systems require low average latency (delivery delay). However, true real-time systems succeed only if they always respond "on time." This is the maximum latency, often expressed as the average delay plus the variation or "jitter." Even a fast server with low average latency can experience large jitter under load. RTI Connext DDS achieves real-time performance by sending information directly between peers with no servers or brokers, by using multicast when possible, and by optimizing both send and receive code paths.

When running over an unreliable medium like Ethernet, no software solution can guarantee deterministic updates to remote recipients. RTI nonetheless supports many applications that "require" isochronous (on time) delivery.

Usually, DDS applications do this by leveraging the fact that DDS can notify them locally of any missed data. So, they implement an intelligent "fallback" scheme that provides sufficient reliability. For instance, a feedback controller that needs data within 30 microseconds may do nothing if it misses one sample, go into a "failsafe mode" (like stopping) if it misses two in a row, and stop altogether if it misses three. These types of strategies are easy and amazingly effective, even in safety-critical applications.

This can leverage hardware support in the future. There is a new version of the IEEE 802.11 networking standard in development called "Time Sensitive Networking" or TSN. TSN will allow software to reserve bandwidth



Figure 28. Real-Time Applications. RTI middleware is used in many systems that require controlled latency. Examples include navy weapons systems that must track and shoot down incoming missiles, medical robotics with 3 kHz sample loops, computed tomography imagers with extremely fast X-ray and servo control, and even multichannel real-time video switches.

This works because discovery is handled before the dataflow. During operation, DDS knows exactly where it must send every update. Thus, it can use fast peer-peer transmission.

and timeslots for particular flows and thus guarantee isochronous (on time) delivery. TSN has great potential but is still unfinished and faces challenges. The central configuration, for instance, gets hard for larger systems and imposes many restrictions.

Interestingly, the DDS standard already specifies collection of all the information needed to generate TSN configurations automatically: data type sizes and rates, QoS needs for every participant, and even latency budgets and transport priorities. Thus, it could uniquely take advantage of TSN without complex, static, manual configurations. Even better, using TSN with DDS maintains the global data abstraction, and that's the real benefit.

Of course, TSN is neither always justified nor a panacea. For instance, as "regular" networking gets faster, TSN is less necessary. There's little reason, for instance, to doubt reliability for a 100Hz controller over a dedicated, lightly-loaded 1Gbit Ethernet with real-time operating systems. And, TSN cannot compensate for other factors like operating system or network stack variances; these are often bigger problems than the hardware transport. Its most appealing use is to simplify adding applications and options without threatening reliability. Thus, developers should realize both the costs and benefits.

TSN also offers the potential to push smart distributed software systems deeper into the formerly-exclusive domain of isochronous hardware like fieldbuses. This can be useful in autonomy. However, as of this writing, TSN is still an evolving technology. We expect TSN to be useful for many, but not for every application. A fallback scheme is simpler and nearly always important, because even an isochronous underlying transport can fail. So, developers should carefully consider their real requirements before jumping to the assumption that they need the complexity of a specialized TSN transport.

EASE SYSTEM INTEGRATION

Intelligent systems manage many variables and very complex state relationships. A DDS databus excels at decoupling the complexity.

Large systems contain many "modules," each module is typically an independently-developed application built by an independent team of developers. Module scale quickly



Figure 29. Challenging System Integration. Modern cars, like many intelligent systems, must integrate the work of many teams. The DDG-1000 ship, for instance, integrates thousands of software modules built by 1500 teams of programmers. Audi's Hardware-In-the-Loop simulation system must support the development of hundreds of ECUs. Hospital systems integrate hundreds of types of devices. Data-centric technologies directly manage the interfaces between these teams, greatly easing system integration.

becomes a key architectural driver. The reason: system integration is inherently an "n-squared" problem. Each new team presents another interface into the system. Smaller projects built by a cohesive team can easily share interface specifications without formality. Larger projects built by many independent groups of developers face a daunting challenge. System integration can occupy half of the delivery schedule and most of its risk.

In these large systems, interface control dominates the interoperability challenge. It is not practical to expect interfaces to be static. Modules or groups of modules that depend on an evolving interface schema must somehow continue to interoperate with older versions of that schema. Communicating all the interfaces becomes hard. Forcing all modules to "update" on a coordinated timeframe to a new schema becomes impossible. Thus, interacting teams quickly find they need tools, processes and eventually architectural support to solve the system integration problem.

Of course, this is a well-studied problem in enterprise software systems. Data-centric systems expose and control interfaces directly, thus easing system integration. In the storage world, databases ease system integration by explicitly modeling and controlling "data tables," thus allowing multiple applications to access information in a controlled manner. However, databases provide storage for data at rest. Most IIoT systems require data in motion, not (or in addition to) data at rest.

Data-centric middleware is a relatively new concept for distributed systems. Similar to a database data table, data-centric middleware allows applications to interact

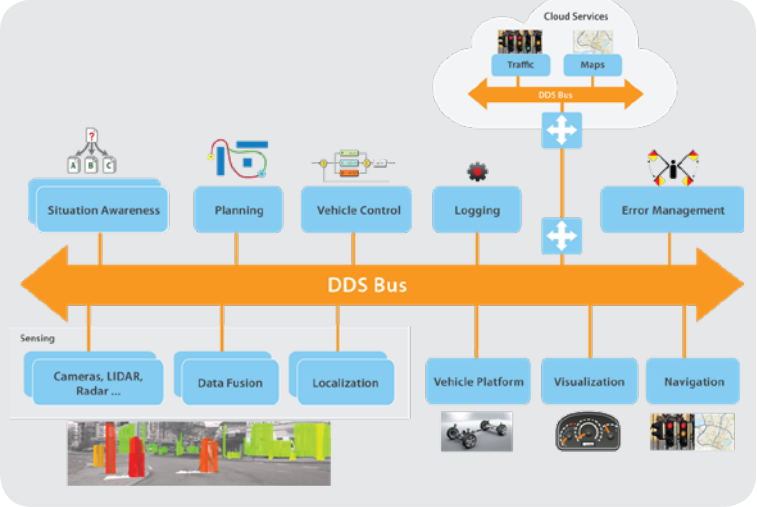


Figure 30. Autonomous Car System Architecture. DDS integrates all the components in a typical autonomous car design. The data-centric interface means that all module interactions are strictly controlled for schema matching, rates, reliability and system health. Each of these components is a complex module on its own. Powerful system integration support lets teams work independently with the assurance that the infrastructure will directly support the data interaction needed for system-wide operation.

through explicit data models. Advanced technologies can even detect and manage differences in interfaces between modules, then adapt to deliver to each endpoint in the schema that the endpoint expects. These systems thus decouple application interface dependencies, allowing large projects to evolve interfaces and make parallel progress on multiple fronts.

BUILD SECURITY IN FROM THE START

Security is critical. To make security work correctly, it must be intimately married to the architecture. For instance, the "core" standard may support various patterns and delivery capabilities. The security design must match those exactly. For example, if the connectivity supports publish/subscribe, so must security. If the core supports multicast, so must security. If the core supports dynamic plug-and-play discovery, so must security. Security that is this intimately married to the architecture can be imposed at any time without changing the code. Security becomes just another controlled quality of service, albeit more complexly configured. This is a very powerful concept.

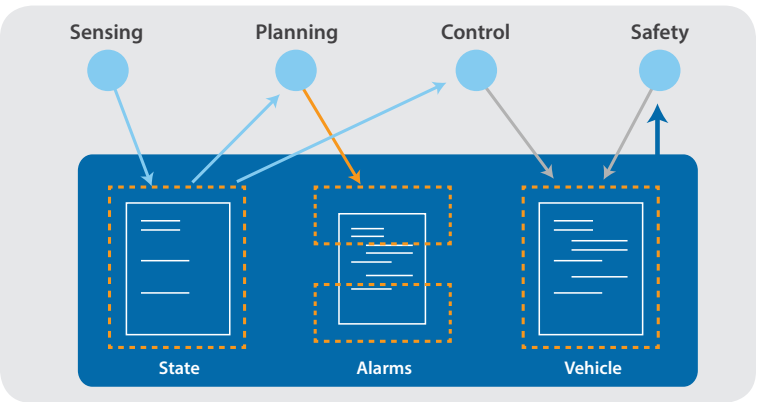


Figure 31. Dataflow-Level Security. The DDS standard security model supports all the key security components: authentication, access control, encryption, tagging and logging, and even non-repudiation. The secured elements are not "users" or "pipes," but actual system modules. A simple configuration controls which modules can read and write which data items. Thus, the security architecture exactly matches the connectivity architecture.

In the example figure, the sensing module can only affect state. Planning can read state and set alarms. Control can read state and affect the vehicle. Safety can read anything and control the vehicle. This level of security limits the impact of a breach of any one module.

MAKE DEPLOYMENT FLEXIBLE

Data-centric design implies that only data interactions matter. Thus, modules may run anywhere in the system. There can even be redundant copies of modules running for reliability. Location, chip architecture, language and operating system differences are completely transparent.

Data centricity very much simplifies practical deployment of mobile systems as well. RTI now supports "IP mobility," so software components are not dependent on their physical position or network connectivity. For instance, this lets a carbot move between cell towers without losing reliable dataflow or having to renegotiate security certificates. Applications communicate with data, not with entities with a fixed IP address. This extends the global data space abstraction to moving, changing systems.

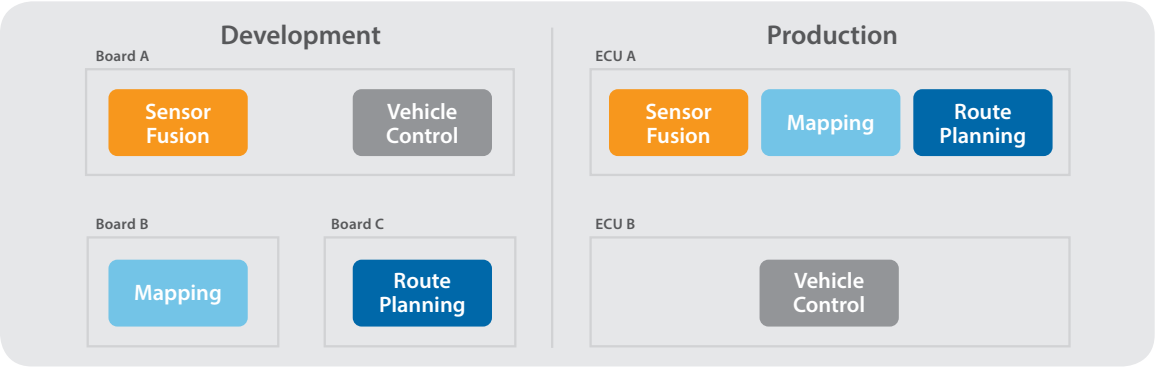


Figure 32. Deployment Flexibility. Data centricity separates physical location, transport, chip architecture, language and operating system from operation. Thus, system designers can move modules between physical locations. As long as the data relationships are satisfied, the system will work. Configurations can change as teams move through evaluation, prototyping, testing, production and maintenance.

EASE SAFETY CERTIFICATION

DDS is already running many safety-critical systems. One of the most demanding is a system called Ground Based Sense and Avoid (GBSAA), which allows autonomous planes to fly in the US National Air Space (NAS). As part of getting this system running, RTI put a version of its Connext DDS product through full DO-178C-level A certification, the FAA's most stringent standard. Certification evidence includes 940 High-Level Requirements, 3,680 Low-Level Requirements, 3,400 test files (many of which have multiple test cases), and 99.88% code coverage testing.

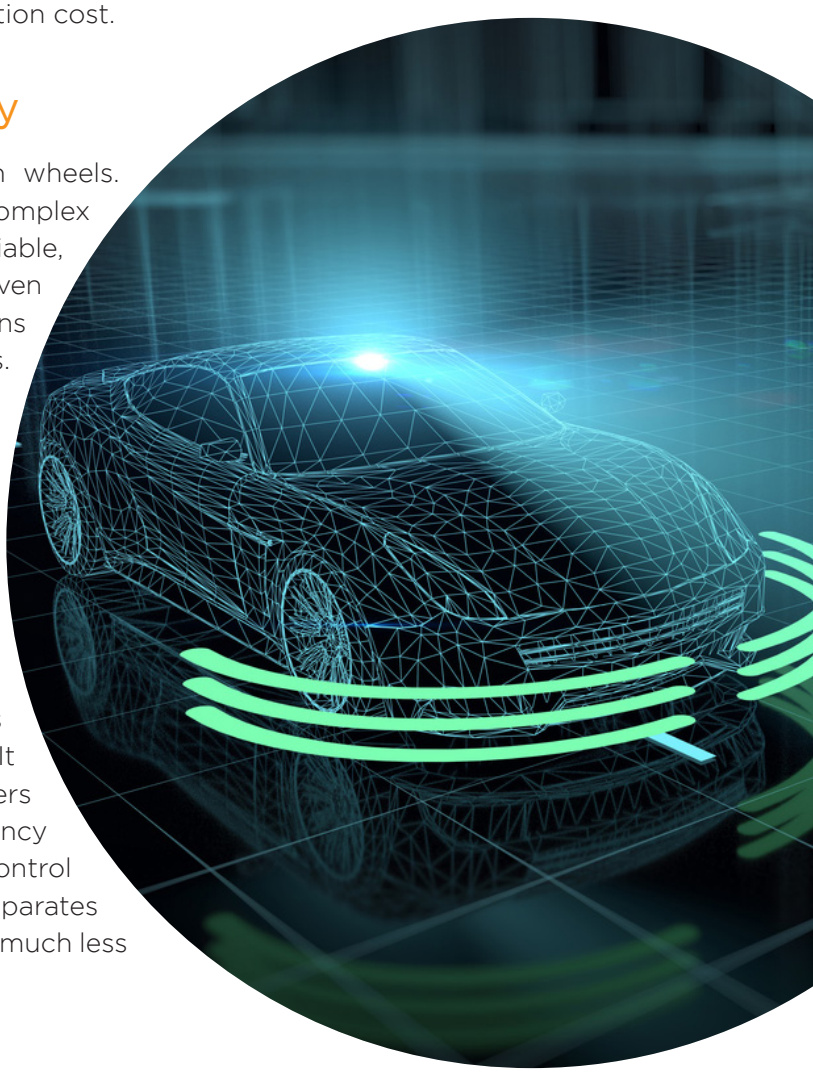
DO-178C-level A is more demanding than the most important standard affecting autonomous cars, the ISO 26262 road-vehicle functional safety specification. Thus, RTI's Connext DDS Cert product can directly support vehicles that require safety certification. Importantly, teams can develop with the very functional "full" DDS, and then deploy critical modules with fully certifiable middleware.

Certified middleware also separates user code modules. As an isolating layer, certified middleware divides the system into provably separate modules. Thus, the system can easily combine critical components with non-critical subsystems. RTI's decentralized support for redundancy, automatic failover, serverless operation and evolutionary upgrades makes expensive certified systems much more modular and therefore much less expensive to certify. Because the much-smaller certified version interoperates with the "full" version, developers can mix-and-match functionality without increasing certification cost.

Carbots and Connectivity

An autonomous vehicle is a robot on wheels. Connectivity is key to success for complex robotic systems. The databus delivers reliable, flexible, real-time, secure connectivity proven in operational autonomous applications and many product lines across industries.

With extensive QoS filtering and delivery control, DDS implements an extremely optimized system. It adapts to all the varying components, including perception, mapping and navigation, connection off vehicle, decision and planning, display and visualization, and vehicle control. It can handle high-volume video and sensors as easily as low-latency feedback. It manages redundancy naturally; developers can implement double or triple redundancy with no extra work. It adds security control to all dataflows. It also provably separates modules, making certification easier and much less expensive.



Thus, DDS delivers microsecond latency, easy scalability, IEC 26262 safety certification and top security. Because it separates and controls module interfaces directly, DDS eases development, integration, debugging and deployment.

Systems work by sharing data. Real-time systems work by controlling that shared data to effect reliable navigation through real-world challenges. And smart systems need to intimately marry the dataflow to powerful computing.

The databus and its global dataspace abstraction makes this much more practical. It greatly simplifies redundancy to increase reliability. It allows teams to divide their work, yet still interact reliably in real time. And, it eases deployment by making physical location flexible, by integrating security and by easing certification.

The imminent practicality of carbots fulfils my early ambition to make transportation safer for all. And, carbots create entirely new economic, mobility and application opportunities. They are a classic example of technology protecting us from our own folly while giving us back countless hours lost to driving and traffic. We really need carbots.

THE AGE OF THE ROBOT OVERLORDS

Historians will look back at this time and wonder how we got by without smart things. Medicine is expensive, slow and error-prone. We still burn things for power and, in turn, damage the planet. Large systems, from process plants to defense, are far too expensive to develop. And transportation is slow and dangerous. What a waste! Who needs traffic and accidents and pollution? Things can be so much better. Why doesn't someone do something about it?

Someone, or rather, something, is: the robot overlords.

Historians will also look at the next two or three decades and see that the most important technology driver is the rise of artificial intelligence and pervasive device

networking. Soon, essentially everything humanmade will be both smart and connected. That may sound easy, and maybe it is for thermostats and Fitbits. But what about things that monitor lives, protect safety and drive cars? Those things have to work: every application, every place, every time. That is anything but easy. But it is on the horizon.

The robot overlords aren't evil or controlling. The robot overlords, in fact, are just the next step in technology that has improved our lives since the invention of fire. That next step is to connect smarts to things. The trick is to do it reliably enough, fast enough and cheaply enough to run our most critical industries and protect our most precious assets. That is the legacy of the Industrial IoT. We have never before had the power to improve our world as dramatically as we do today.

I can't wait to be a pet.

ABOUT RTI

Real-Time Innovations (RTI) is the Industrial Internet of Things (IIoT) connectivity company. The RTI Connext® databus is a software framework that shares information in real time, making applications work together as one, integrated system. It connects across field, fog and cloud. Its reliability, security, performance and scalability are proven in the most demanding industrial systems. Deployed systems include medical devices and imaging; wind, hydro and solar power; autonomous planes, trains and cars; traffic control; Oil and Gas; robotics, ships and defense.

RTI lives at the intersection of functional artificial intelligence and pervasive networkingSM.

RTI is the largest vendor of products based on the Object Management Group (OMG) Data Distribution Service™ (DDS) standard. RTI is privately held and headquartered in Sunnyvale, Calif.



Your Systems. Working as One.

CORPORATE HEADQUARTERS
232 E. Java Drive
Sunnyvale, CA 94089
Tel: +1 (408) 990-7400
Fax: +1 (408) 990-7402
info@rti.com
www.rti.com

RTI, Real-Time Innovations, RTI Data Distribution Service, Connext, Micro DDS, 1RTI, and the phrase "Your systems. Working as one," are registered trademarks or trademarks of Real-Time Innovations, Inc. All other trademarks used in this document are the property of their respective owners. ©2018 RTI. All rights reserved. v. 90001 0618