APPLYING DDS TO THE GAMING FLOOR

WHITE PAPER

July 28, 2019

GAMING STANDARDS ASSOCIATION

Introduction

Communications on gaming floors continues to challenge the industry. Managing communications between thousands of gaming machines and many disparate systems has proven to be a difficult and complex task. The G2S protocol was one attempt at a solution to this problem. However, the level of effort required to implement the core communications, security, and data delivery requirements has hampered adoption.

This paper looks at an alternative approach to communications, security, and data delivery – Data Distribution Service (DDS). It is a middleware protocol and API for data-centric connectivity managed by the Object Management Group (OMG). DDS handles communications, security, and data delivery allowing developers to focus internally on their applications rather than externally on communications with other endpoints. It isolates the applications from the complexities of communications, security, and data delivery. They can change and evolve without impacting the applications.

This paper provides an overview of DDS, explaining what DDS is and how it works, and then looks at how G2S-like functionality could be implemented using DDS. Much of the information about DDS was borrowed from OMG's website (www.dds-foundation.org). Further information about DDS can be found on that site.

What Are the Benefits of DDS?

Adoption of DDS as the middleware for communications on casino floors would result in a series of benefits for the gaming industry. Most of these benefits result from using off-the-shelf middleware for communications, security, and data access rather than proprietary industry-specific middleware. Secondary benefits come from the native communications patterns built into DDS – publish-subscribe and request-response – which do not have to be built into applications.

- **Faster Development.** Suppliers would not have to worry about communications, security, and data access or managing the publish-subscribe and request-response communications patterns.

- **Focus on Applications.** Suppliers could focus on their applications rather than the infrastructure needed to communicate with hundreds or thousands of nodes on a gaming floor.

- **Independent Control**. Security and data access would be managed independently of the products on the gaming floor, placing management of the gaming floor in the hands of the operator.

- **Unfettered Access.** Data access would be controlled through DDS rather than proprietary systems, giving operators complete control of the data.

- **More Data.** Data would be delivered by DDS rather than proprietary systems, giving operators access to the full spectrum of data offered by EGMs.

- **Open Gaming Floor.** The infrastructure surrounding communications, security, and data access would not be a barrier-to-entry to the gaming floor, giving operators more choice and making in-house development easier.

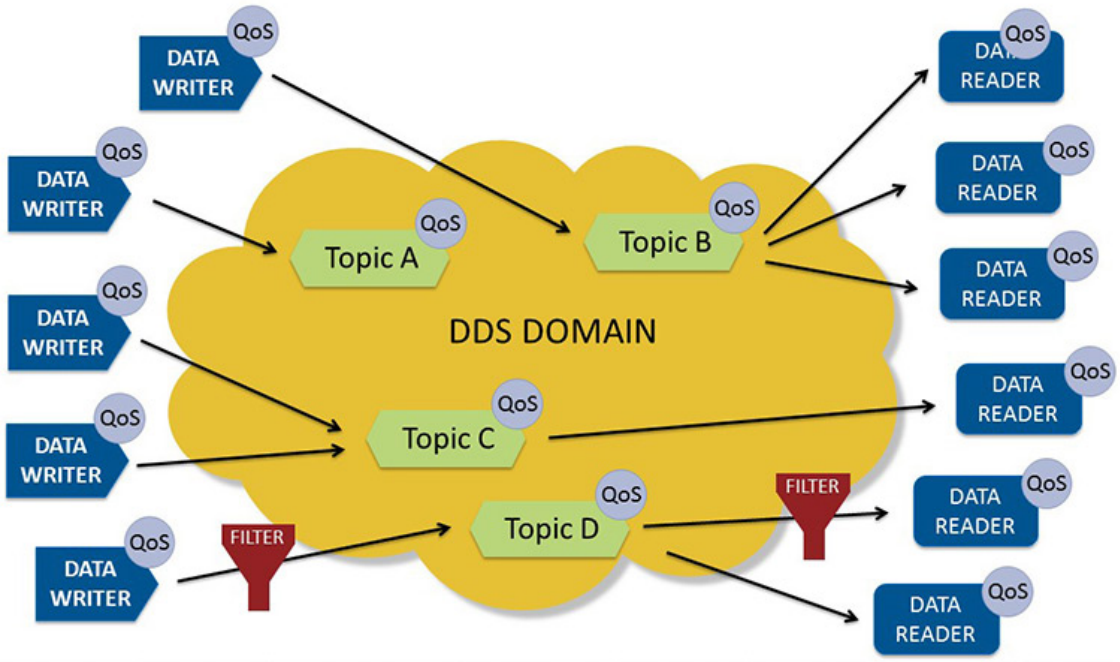
- **Level Playing Field.** No supplier would have any advantage over any other supplier due to control of the infrastructure on the gaming floor.

The net result would be lower development costs, faster speed-to-market, fewer barriers-to-entry, more product availability, and increased revenues for suppliers and operators.

What Is DDS?

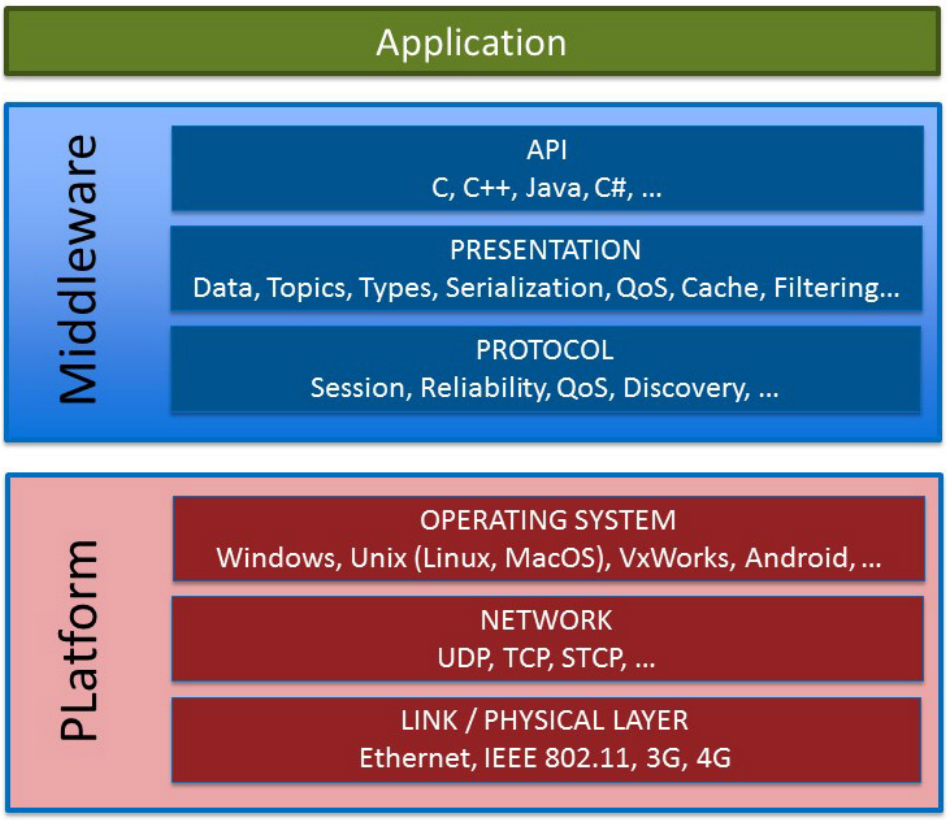
DDS integrates the components of a system together, providing low-latency data connectivity, extreme reliability, and a scalable architecture that is needed by business and mission-critical applications.

DDS is the software layer that lies between the operating system and the applications. It allows the various components of a system to more easily communicate and share data. It simplifies the development of distributed systems by letting software developers focus on the specific purpose of their applications rather than the mechanics of passing information between applications and systems.



DDS provides QoS-controlled data-sharing. Applications communicate by publishing and subscribing to Topics. Subscribers can specify time and content filters and, thus, get only a subset of the data being published on a Topic and only when it is needed.

There are many communications middleware standards and products. DDS is uniquely data centric. DDS knows what data is stored in Data Writers and controls how to share that data with Data Readers. There is no broker or central data distribution system like those used with other technologies, such as message queues. DDS is a purely distributed system.



Conceptually, DDS looks like a local data store called the global data space. To the application, the global data space looks like native memory. It is accessed via a local API. You write to what looks like your local storage. In the background, DDS sends messages to update the appropriate data stores on remote nodes. The remote nodes read from what looks like local storage.

Each node stores only what it needs and only for as long as it needs it. DDS deals with data in motion. The global data space is a virtual concept; in reality, it is only a collection of local stores. Every application sees the data in local memory in its optimal native format. DDS shares data between embedded, mobile, and cloud applications across transports, regardless of language or operating system, and with extremely low latency.

DDS provides dynamic discovery of Data Writers and Data Readers. Endpoints do not have to be configured into applications. They are automatically discovered by DDS.

This is completed at run time, not at design or compile time, enabling true "plug-and-play" for DDS applications.

DDS will discover if an endpoint is publishing data, subscribing to data, or both. It will discover the Topics being published or subscribed to. It will also discover the publisher's offered communication characteristics and the subscriber's requested communications characteristics, using these attributes during the dynamic discovery to match DDS participants.

Quality of Service (QoS) constraints include reliability, durability, timeliness, and liveliness (system health). Every endpoint does not need every item in the global data space. DDS is smart about sending just what it is needed when it is needed. If messages don't reach their intended destinations, DDS enforces the reliability constraints for the Topic. When systems change, DDS dynamically figures out where to send which data and intelligently informs participants of the changes. When updates need to be fast, DDS sends multicast messages to update many remote endpoints at once. Use of multicast groups allows routers to only forward multicast messages to endpoints that have subscriptions to the groups. When security is needed, DDS limits access to Topics and encrypts data on-the-fly.

DDS was developed for the Internet of Things (IoT). It was designed to manage communications and data delivery amongst many thousands of endpoints in real-time with very low latency. It is used by many industries developing mission-critical and business-critical applications. These applications include air traffic control, smart cars, power distribution, medical devices, industrial automation, launch control systems, and combat management systems.

OMG is supported by approximately 350 suppliers and users of products that incorporate OMG's standards. These include AT&T, Bloomberg, Boeing, Ford, General Dynamic, General Electric, Goldman Sachs, HP, IBM, Microsoft, MITRE, NASA, Oracle, SAP, and Thales.

How Does DDS Work?

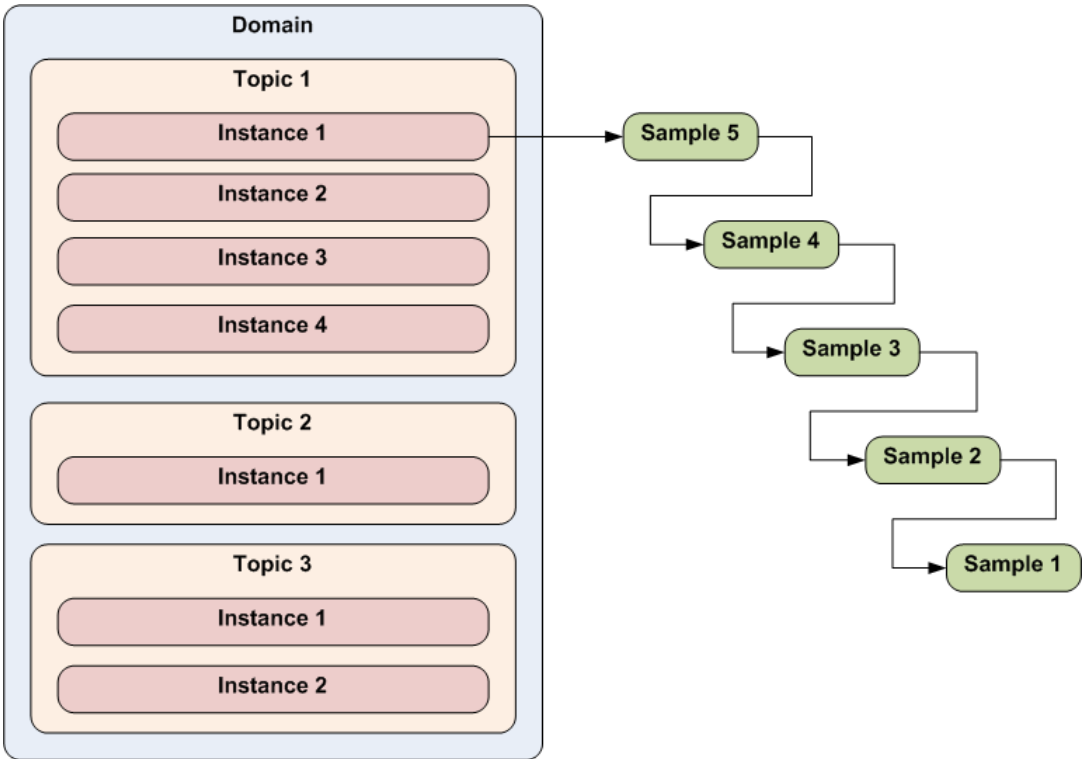
Conceptually, DDS is quite simple. It relies on a small handful of core concepts.

- **Domain** – A Domain is the global data space. Different Domains are completely independent from each other. There is no data-sharing across domains. A Topic in one Domain may represent something completely different in another Domain, even if they share the same Name. However, an application can participate in multiple domains. Domains can be sub-divided into Partitions when necessary.

- **Topic** – Participants publish and subscribe to Topics. Data Writers publish data to Topics by writing the data to local data stores. Data Readers receive data from Topics by reading from local data stores. DDS moves data between the local data stores. Each Topic has a Name, Data Type, and QoS. The Data Type can be an XML element, a JSON object, or some other type of data structure. They can be quite large and complex when necessary or as simple a single value. The QoS defines the reliability, durability, timeliness, and liveliness requirements for the Topic.

- **Instance** – An Instance is an occurrence of a Topic. There can be multiples Instances of a Topic within a Domain and multiple Instances on a participant. Participants create, update, and delete instances as needed. It is possible to restrict a Topic to a single Instance; this may be useful for setting global parameters. When multiple instances are allowed, parameters within the Data Type are formed into a unique Key for Instances of the Topic. Subscribers to a Topic are alerted by DDS when Instances are created, updated, or removed. The data associated with a new Instance automatically becomes part of the subscriber's local data store.

- **Sample** – A Sample is a snapshot of the data within an Instance. Samples are communicated between Data Writers and Data Readers. The local data store of a participant may contain multiple Samples of an Instance. For example, a local data store could contain two samples of meter information – the current and previous meter readings.



It's easy to draw an analogy between these concepts and a relational database.

- The Domain is equivalent to the database.
- A Topic is equivalent to a table in the database.
- An Instance is equivalent to a record in the table.
- A Sample is a copy of the record at a point in time.

For example, GSA could define a "gaming" Domain (the database). In that Domain, GSA could also define a "meters" Topic (a table) to report meter information from

Electronic Gaming Machines (EGMs). The unique Key for the "meters" Topic could be the EGM Identifier, Class Identifier, and Device Identifier. In practice, there would be multiple Instances (records) of the "meters" Topic – one Instance for each class and device in an EGM. These could be created in real-time as needed. When the contents of an Instance changed, a Sample (copy) of the Instance would be communicated between the Data Writer for the Instance and the Data Readers.

What Is Quality of Service (QoS)?

QoS is a very important part of DDS. It controls the delivery of data between participants. Amongst other things, it defines the required reliability, durability, timeliness, and liveliness behavior for a Topic. DDS takes care of these problems, not the application. In total, there are over 20 parameters involved. These are called policies. Important policies include:

- **Reliability** – The Reliability policy determines whether guaranteed or best-effort delivery is used between Data Writers and Data Readers.
- **History** – The History policy controls how many Samples of an Instance are retained by DDS. Two options are available "keep all" or "keep last". When "keep last" is used, a depth value is also specified – for example, the last 10 Samples. The "keep all" option does not imply that DDS will store Samples indefinitely. The Resource Limits and Lifespan policies also come into play.
- **Resource Limits** – The Resource Limits policy configures the amount of memory a Data Writer or Data Reader may allocate to its local data store for a Topic.
- **Lifespan** – The Lifespan policy specifies how long DDS should consider a Sample to be valid. Once a Sample is no longer considered valid, it is discarded.
- **Deadline** - The Deadline policy specifies the maximum elapsed time between Samples from a Data Writer – that is, the frequency at which data is published.
- **Time-Based Filter** – The Time-Based Filter policy sets the minimum elapsed time before a new Sample is provided to a Data Reader – that is, the frequency at which data is received. Excess Samples are discarded or not sent.
- **Content Filter** – The Content Filter policy contains predicates (the equivalent of SQL "where" clauses) that are applied to Samples. Only Samples that satisfy the policy are delivered to Data Readers.
- **Liveliness** – The Liveliness policy specifies the mechanism that will be used by Data Readers to detect when Data Writers become disconnected or dead. Two options are available: automatic or manual. Automatic is a simple ping mechanism between the Data Readers and Data Writers. Manual requires that the Data Writer application initiate the ping and that the Data Reader application respond.

- **Partition** – The Partition policy adds additional requirements for matching Data Writers to Data Readers for a Topic, allowing a Domain to be divided into subsets. Date Writers are only matched to Data Readers if their Partitions intersect – for example, if an EGM is participating in a specific progressive jackpot. Different Partitions can be associated with different multicast groups so that routers only send data to interested Data Readers.

For a Data Reader to be matched by DDS with a Data Writer, the Name and Data Type of the Topics must be the same. In addition, the QoS offered by the Data Writer must be greater than or equal to the QoS requested by the Data Reader. For example, if the Data Writer offers best-effort Reliability but the Data Reader requests guaranteed Reliability, the Data Writer and Data Reader will not be matched; best-effort delivery is a lower QoS than guaranteed delivery. Once a Data Reader has been matched with a Data Writer, DDS will manage the delivery of Samples to the Data Reader per the QoS policies of the Data Reader.

Over time, new versions of Topics can be introduced to a Domain. This can be accomplished by introducing new Data Types for a Topic or by simply using the native extensibility model for the underlying data structures to extend the Data Type for a Topic.

When developing a DDS-based solution, the key role for a standards-setting organization, such as GSA, is in the definition of the Domains, Topics, Data Types, and QoS policies for Data Writers of the Topics. The focus is on the data; it is data-centric, allowing easy integration between endpoints. This is very much like the work that GSA does today defining the data models used in its other standards, such as G2S.

What Patterns Are Supported by DDS?

Two of the communications patterns supported by DDS – publish-subscribe and request-response – would be ideally suited for transitioning a gaming floor to DDS. These patterns are similar to patterns currently used in G2S.

- **Publish-Subscribe** – Publish-subscribe is an ideal way to distribute meters, events, status, configuration, and log information from EGMs to systems. EGMs simply need to expose the appropriate Data Writers, updating their local data stores when necessary. Systems create corresponding Data Readers, with appropriate Time-Based and Content Filters, and then read the information from their local data stores as it becomes available. DDS handles the distribution of the data to the Data Readers.

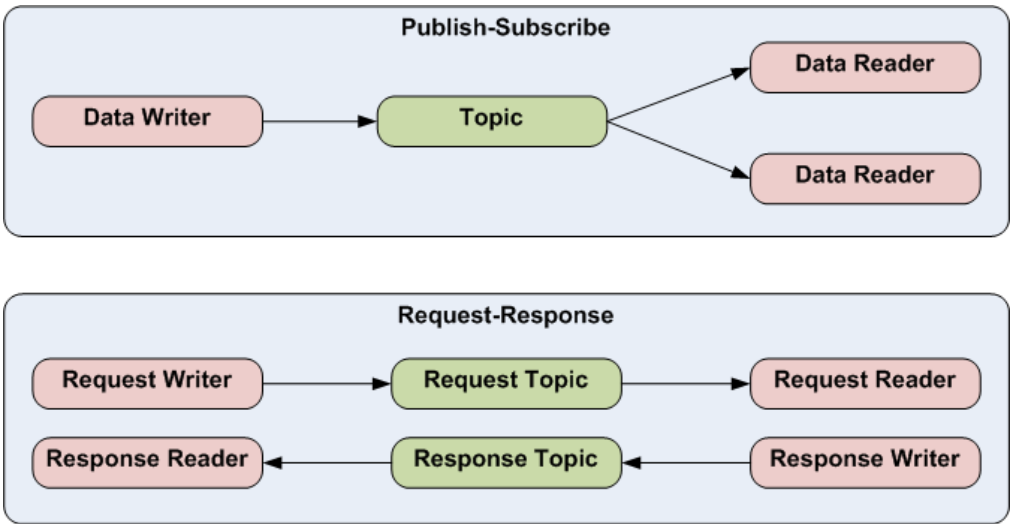
- **Request-Response** – Request-Response is an ideal way to manage remote configuration, software download, and monetary transactions. Clients simply need to write their requests to the appropriate Request Topic. DDS RPC will deliver the requests to the appropriate servers and bring back their replies, delivering the replies to the local data store of the clients.

Publish-subscribe is the simpler pattern. It is achieved by simply writing Samples to a Topic. The Data Writer publishes the Samples to the Topic. DDS delivers the

Sample to any Data Readers that have subscribed to the Topic, applying Time-Based Filters and Content Filters along the way. The Samples appear in the local data store of the Data Readers.

Meta-information tells the application which Samples have been read by the application (Sample State) and whether a Sample is a new Instance or an update (View State). The application can either read or take the Samples. Taking removes the Samples from the local data store; reading leaves the Samples in the local data store. If not taken by the application, policies such as History, Resource Limits, and Lifespan control how long the Samples will remain in the local data store of the Data Reader.

The request-response pattern is built on top of the publish-subscribe pattern. Under the covers, there are two Topics: the Request Topic and the Response Topic. The client, which is making the request, publishes the request to the Request Topic. The request is delivered by DDS to the server, which processes the request. The server publishes its reply to the Response Topic. DDS delivers the reply back to the client.



Unfortunately, the underlying publish-subscribe pattern, does not address all problems that can arise when using the request-response pattern. For example, the publish-subscribe pattern by itself does not address message correlation (matching replies to requests), exception handling, and time-outs. Solutions to these problems need to be built into the application layer. Alternatively, DDS RPC can be used.

DDS RPC provides a standard approach to implementing the request-response pattern on top of the publish-subscribe pattern. By introducing a standard methodology, DDS RPC shields applications from many of the complexities of the request-response pattern, helping to simplify implementations and leading to more consistent implementations of the publish-subscribe pattern.

Can You Give Me an Example?

Let's look at how the functionality in the G2S voucher class (ticket-in-ticket-out) would be implemented in DDS. The G2S voucher class is a good example because it touches upon all the standard building blocks of G2S.

- **Profile** – The current configuration of a device.
- **Status** – The current state of a device.
- **Meters** – The current values of the meters for a device and class (all devices in total).
- **Logs** – The transaction history for a class (all devices).
- **Events** – Changes to the profile, status, meters, and logs.
- **Commands** – EGM-originated and host-originated requests.

In DDS, each of these building blocks would become a Topic or RPC. The Data Types for the Topics and RPCs would be very similar to the XML data structures used in G2S (in some cases, they might be identical).

- **voucherProfile** – There would be one Instance of this Topic for each voucher device. Every time there was a change to the profile of a device, a new Sample would be published to DDS. The unique key for the Topic would be EGM Identifier, Device Class, and Device Identifier.

- **voucherStatus** – Like the profile, there would be one Instance of this Topic for each voucher device. Every time there was a change to the status of a device, a new Sample would be published to DDS. The unique key for the Topic would be EGM Identifier, Device Class, and Device Identifier.

- **voucherMeters** – There would be multiple Instances of this Topic – one Instance for each voucher device and one Instance for the voucher class in total. Every time there was a change to the meters for a device, a new Sample would be published to DDS for the device and for the class in total. The unique key for the Topic would be EGM Identifier, Device Class, and Device Identifier. As in G2S, Device Identifier 0 (zero) would identify the meters for the class in total.

- **voucherLog** – There would be multiple Instances of this Topic – one Instance for each transaction in the voucher class. Every time a new transaction was started, a new Instance would be created and then a Sample would be published to DDS. Subsequently, as the state of the transaction changed, the Instance would be updated, and then additional Samples would be published. The unique key for the Topic would be EGM Identifier and Transaction Identifier.

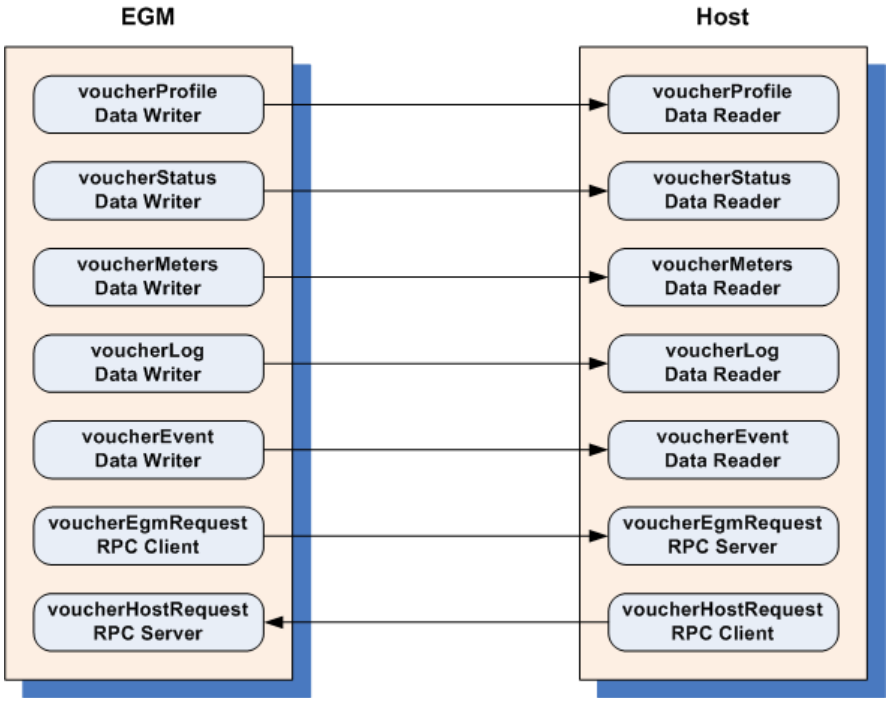
- **voucherEvent** – Like the log, there would be multiple Instances of this Topic – one Instance for each event in the voucher class. Every time an event was generated, a new Instance would be created and then a Sample would be published to DDS. The unique key for the Topic would be EGM Identifier and Event Identifier.

- **voucherEgmRequest** – This RPC would be used by EGMs (acting as clients) to send requests to a host (acting as the server). The requests would be

equivalent to the G2S getValidationData, issueVoucher, redeemVoucher, and commitVoucher commands.

- **voucherHostRequest** – This RPC would be used by a host (acting as the client) to send requests to EGMs (acting as servers). The requests would be equivalent to the G2S setVoucherState and setVoucherLockOut commands.

The QoS for the Topics would be designed to match the requirements of G2S. In general, guaranteed delivery, as well as data persistence, would be required for all Topics. The History policy for logs and events would be set to depths equivalent to those used by G2S.



How Is DDS Secured?

DDS Security is an add-on to DDS in much the same way that TLS is an add-on to HTTP. DDS Security can be added at any time to secure some or all the Topics in a Domain. It plugs into DDS. Applications can be fully developed without DDS Security in place. DDS Security can be added and configured in the field as necessary. Security is part of DDS, not the applications.

DDS Security uses a standard Public Key Infrastructure (PKI) and X.509 v3 certificates to identify and authenticate the participants in a Domain. A trusted Certificate Authority is used to sign, authenticate, and revoke the certificates of participants. This is very similar to the security model used by G2S.

On top of this, DDS Security uses two XML-based documents to control access to Topics.

- **Domain Governance Document** – The Domain Governance Document contains the global set of rules for the for the Domain. All participants must abide by these rules.

- **Participant Permissions Document** – The Participant Permissions Document contains the participant's permissions for the domain, allowing the participant to publish or subscribe to specific Topics.

Similar participants, such as EGMs, can share a common set of permissions. Alternatively, a different set of permissions can be constructed for each participant. In a gaming network, each server would probably have its own set of permissions tuned to its specific functional needs while EGMs would share a common set of permissions based on operator and jurisdictional needs.

A trusted Permissions Authority is used to sign the Domain Governance and Participant Permissions Documents. These documents are then installed on the participants along with the X.509 v3 certificates that are used to identify and authenticate the participants. To help simplify implementations, the Certificate Authority can also act as the Permissions Authority.

How Is Persistence Achieved?

Data persistence is typically viewed as a service. When needed, the service is called by a Data Writer and the relevant data is stored in persistent memory. Essentially, the service acts as a subscriber to the data, storing the data on disk or other type of persistent media. The service can be local to the participant or it can be remote. The service is responsible for "re-publishing" the data to late-joining subscribers – that is, subscribers that were not active when the data was first published.

Each DDS implementor must decide where to host the service. In theory, a single network-based service could provide persistence services to an entire Domain. However, for fault-tolerance and efficiency, usually the persistence service is hosted locally on each participant.

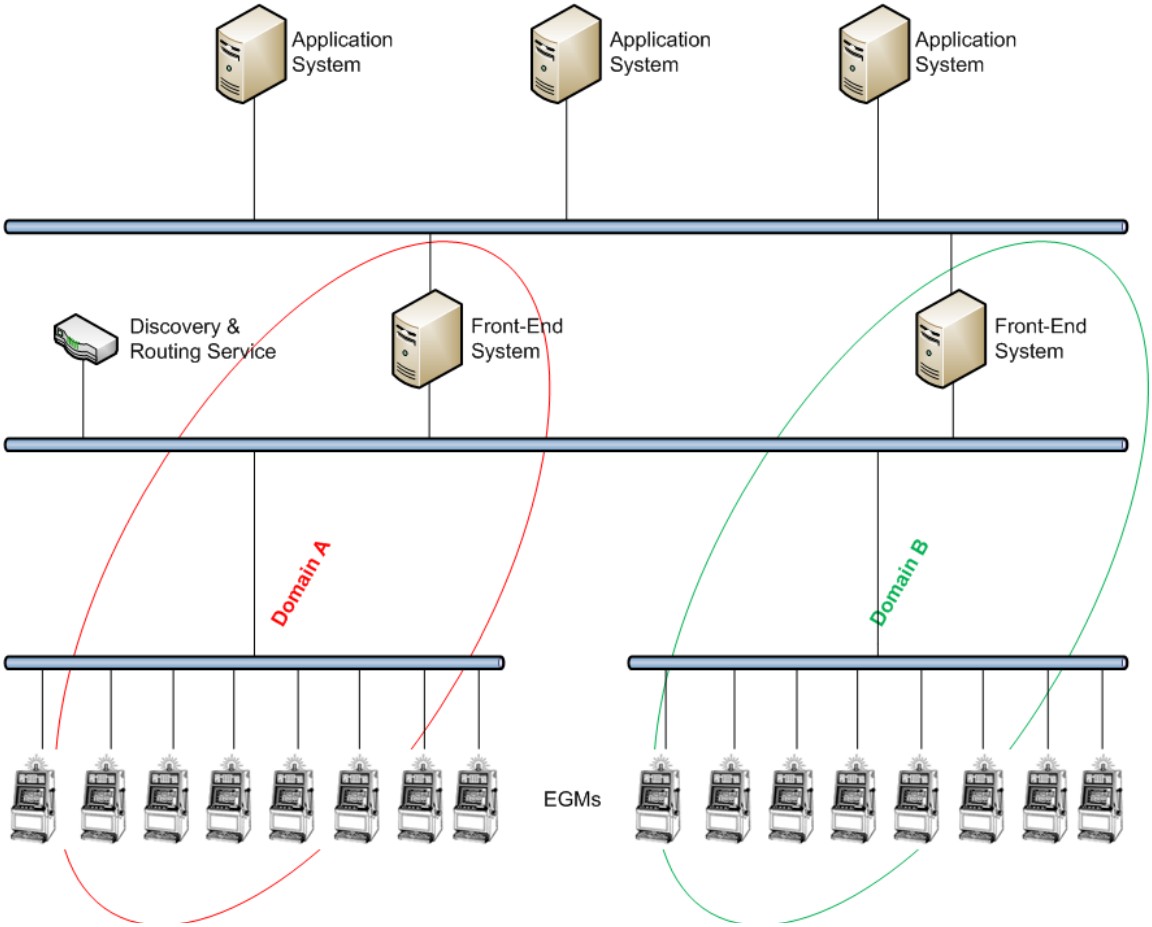
Will DDS Products Interoperate?

DDS is unique in that it standardizes both the API used by the application to access DDS as well as the wire protocol between participants. The standardized API provides portability of applications between DDS suppliers. The standardized wire protocol ensures interoperability between different DDS suppliers.

Other technologies provide a standardized wire protocol, but not a standardized API. DDS provides both. Suppliers compete based on the efficiency and performance of their DDS implementations, not proprietary extensions to the DDS API or the wire protocol.

How Does DDS Scale?

Thousands of Data Writers can easily overwhelm a single Data Reader. Thus, strategies are needed to scale applications as the number of Data Writers grows. Typically, this is done by subdividing a large Domain into a series of smaller sub-Domains and then providing separate front-end systems for each sub-Domain. The front-end systems dispatch critical data and transactions to back-end application systems. This is similar to the strategy used on gaming floors today when site controllers and floor controllers are used to divide the workload across multiple front-end systems.



Other strategies may be available from DDS suppliers to deal with scaling issues, such as load-balancing, discovery, and routing. These are designed to deal with the realities of large networks and to improve the level of fault tolerance in the networks. These are features of the supplier's DDS implementation and, thus, do not impact the DDS API or the wire-protocol.

What Does the Product Landscape Look like?

DDS is a software product. Typically, the DDS software runs natively on the participant – for example, on the EGM or back-end system. In the case of back-end systems, an implementor may choose to run DDS on a dedicated server and then dispatch data and critical transactions to an existing back-end application system. This could be the same application system that is communicating with EGMs that use SAS or G2S.

OMG's website lists a number of DDS suppliers including Adlink (a GSA member), Real-Time Innovations, and Twin Oak Computing. DDS suppliers typically offer two basic types of products – DDS development tools (SDKs) and DDS run-time engines. The tools and run-time engines are available for most popular languages and platforms.

The suppliers may also offer specialized add-on applications that help improve DDS efficiency and performance, such as routing and discovery services. These product offering vary by supplier and may be specific to their DDS implementation.

There are different pricing options available, but typically users of DDS pay for the development tools on an annual basis based on the number of seats required. Pricing of the run-time engines varies from completely free (without support) to a fixed annual price per device, which typically varies depending on the quantity of devices. There may be additional annual fees for add-on products, such as discovery and routing services.

For example, you might expect to pay $5,000 per year per developer for the DDS development tools plus $10 per device for each fully support run-time license (run-time licenses without support are free). An add-on discovery and routing service might cost an additional $3,000 per year.


20
YEARS
1998 - 2018
GAMING STANDARDS
ASSOCIATION