WHITEPAPER

Repeat Success, Not Mistakes;

Use DDS Best Practices to Design Your Complex Distributed Systems

Abstract

RTI Connext™ DDS (Data Distribution Service) is a powerful tool that lets you efficiently build and integrate complex distributed systems like no other technology – if you use it right. Be aware of how to get the most out of DDS and how to avoid common pitfalls when developing your system. RTI has developed Connext DDS best practices over the course of hundreds of customer projects and many years.

This whitepaper covers three important types of best practices: architectural, application design and implementation, and network and QoS configuration.

Introduction: Why Best Practices?

Best practices allow you to use a system that not only works now, but is also future-proof. Future-proof systems can be expanded and changed while continuing to work in new configurations and deployments. In addition, best practices allow developers to build systems that are more scalable to meet both current and future requirements. Lastly, best practices help developers build systems that are high performance and testable. This ensures that systems work properly as they're developed as well as down the line.

Without these best practices a system may still work, but it may be suboptimal. The quality of a system may suffer for a variety of reasons if the system can't scale, can't be upgraded or can't be tested. That being said, not all best practices apply all the time. Some customers may not use particular best practices because they trade one aspect of the system for another, such as trading off extensibility for performance. Understanding the benefits of these best practices helps make smart decisions about the tradeoffs of not following them. There are three categories of best practices:

- **Architectural** - Creating a network data model that meets your requirements
- **Application design and implementation** - Making optimal design decisions within a single application
- **Network configuration and QoS** - Tuning for your system requirements

Architectural Best Practices

For a strong network data model, the architect or developer should determine what data is going to be sent between applications. The architect needs to decide up front which data needs to be distributed in the system. The next step is to design how the data is to be sent. This involves deciding which data streams to send, designing the structure of that data and designing the delivery characteristics. There are three key considerations:

- Examine what state is being represented
- Use typed data
- Use keyed data

Examine What State is Being Represented

Imagine that a data set being sent is the state of a fleet of trucks. The data includes the license plate information, the GPS information, the oil level in each truck, the truck manifest and the maintenance information. Each of these types of data is being updated at different rates, or in some cases, periodically. The maintenance information may be changing after a certain number of miles rather than a certain amount of time.

Since these different aspects of the truck state have different characteristics for when they are sent, they also have different requirements for how the middleware must deliver them. For example, data that is sent rapidly and periodically should be delivered with low latency, and not necessarily reliably. In contrast, data that is sent rarely or aperiodically should be delivered reliably.

Data being sent at different rates, and with different delivery characteristics, will generally be mapped into separate data streams. These data streams are called "Topics" in DDS terminology. Topics that represent the state of a fleet of trucks might include:

- TruckGPS
- TruckMaintenance
- TruckManifest

Issues occur when architects or developers haven't thought through the data model, but have instead put everything into one data stream. In this example, this means that all truck data would be sent at the same time – the oil level, manifest, maintenance information and all other truck data would be sent at the high rate of the GPS information. This may work with a limited amount of data, but even at a small scale it uses unnecessary bandwidth and CPU. In a system with more data, this becomes unworkable.

Use Typed Data

Giving data an actual structure or type instead of sending binary data around the network (unless it's binary data such as images and videos) can be very beneficial. If the data has a structure, it should be mapped to the Connext DDS data structures because the middleware handles discovery and distribution of type information.

This allows developers and system integrators to plug in COTS or custom-built tools to visualize data, to record data, to translate real-time data into a relational database or to perform additional functions on the data that are not known in advance.

When applications use typed data it also enables capabilities beyond what was originally designed. New applications can be developed that choreograph, aggregate, split or mediate data without having to worry about protocol-level details. This allows data to be changed in reasonable and well-organized ways, and systems to be maintained and upgraded gracefully.

In contrast, if an application is designed with data that is opaque to the middleware, the application developer must write the logic to convert the data to and from a network form, to support the required programming languages, and to handle endianness. This additional logic must be tested and maintained. This also prevents COTS tools from interacting with the data. Ultimately, it can increase the cost of evolving and upgrading the system.

Use Keyed Data

If multiple real-world objects are being represented in a system, the system architect should use *key fields* to inform the middleware about those objects. Key fields are fields in your data type that form a unique identifier of a real-world object. If data is keyed, the middleware will recognize that each unique value of key fields represents a unique real-world object.

Circling back to the fleet of trucks example, if a developer wants to represent multiple trucks inside a single data stream, they should add a VIN number to each Topic that was previously designed. They should then mark that VIN number as a key field in DDS. The developer has now told Connext DDS that the VIN number is the unique identifier of some object within the TruckGPS topic. Now, if the middleware sees a new VIN identifier it hasn't seen before, it recognizes that it is a new, unfamiliar truck object. In DDS terminology, these unique real-world objects are referred to as *instances*.

Letting the middleware know there are unique objects in the data stream allows it to keep track of life-cycle information for instance objects. For example, it can alert the user if there is an update from a new Army truck that is unfamiliar. If a new truck is added to the fleet, and publishes the "TruckGPS" Topic, the subscribing application is notified of the position and the fact that this is the first time it has received an update about this truck.

The middleware can also keep track of life-cycle information about whether this instance is alive. For example, perhaps the user has stopped receiving updates about a particular Army truck. The user can be notified that this particular truck instance has become not alive. Applications can monitor these life-cycle events to detect problems such as application errors or network disconnections.

The middleware can also use Quality of Service (QoS) to control behavior per instance. Examples include:

- Alerting the user if an update for a particular instance is delayed using the deadline QoS. For example, if the middleware is configured to expect GPS updates from each truck every two seconds, it can be notified that it did not receive an update from a particular truck within that time.
- Allocating a cache per instance using the history QoS. For example, the user can tell the middleware they would like to keep the last five updates per truck. The middleware will then keep a maximum of five updates per truck, and prevent updates about one truck instance from overwriting updates for another.
- Being configured to have a failover mechanism per instance using the ownership QoS. For example, redundant sensors may be set up to update about a particular truck. When each instance represents an individual truck, these sensors can failover gracefully. One sensor might be the owner (primary DataWriter) of the TruckGPS data for a particular truck. If it fails, another sensor can automatically take its place as the owner of that truck's TruckGPS data. These types of redundancies can be set up for each individual object being monitored.

If the system architect does not design the data to be keyed, the middleware will not understand that there are multiple objects being represented. This leads the application developer to write logic that duplicates what is already in the middleware to detect object life cycles, detect delayed object updates, and to provide failover between updates about an object. If the middleware is not aware that the application is representing multiple objects, it cannot keep a separate logical queue per-object. This forces the application to have longer queues to ensure that updates are not lost for a particular object.

In Figure 1, the user tells the middleware that the user data is representing unique trucks. The VIN field is annotated as a key field. This tells the middleware that the VIN field is the unique identifier for a truck. Multiple key fields of different types can be used.

```
struct TruckGPS
{
        string<18> VIN; //@key
        LatLong position;
        long speed;
        long direction;
};
```

Key	Name	Type
True	VIN	string
	position	LatLong
	speed	long
	direction	long

Figure 1. Using keyed data to represent data-oriented objects within the middleware.

Application Design Best Practices

Application developers are usually responsible for following application design best practices. Many of the considerations about which objects to use, how many to create, and when to create them are choices made by application developers rather than system architects. However, these decisions can sometimes impact the entire distributed system. The most common application design best practices are:

- Create few domain participants
- Create entities early
- Never block a callback
- Use WaitSets, not listeners

Create Few Domain Participants

Domain participants are the first objects created in an application when setting up DDS communications. The DomainParticipant object is responsible for setting up communication with other DDS applications within a DDS domain. DDS domains are logically separated communication planes identified by unique numbers. DDS communication does not cross domain boundaries, making domains useful for separating logical subsystems and for testing.

DomainParticipants are responsible for:

- Performing automatic discovery. Applications in the network automatically discover each other and communicate using a standardized discovery protocol. When a DomainParticipant is enabled in an application, it tries to discover other domain participants.
- Maintaining an internal cache containing other DomainParticipants that it has discovered.
- Creating multiple threads for handling different middleware events and receiving data.

More memory is used when there are more DomainParticipants in a system. Every time a DomainParticipant is created, every other application discovers it – this means that creating a DomainParticipant uses memory in both the local application and in all remote applications that discover it. DomainParticipants also consume network bandwidth to discover each other.

The most common mistake that developers make is to create one DomainParticipant per DataReader or DataWriter. This is not required. In general, an application creates only one DomainParticipant for each domain it will join.

Create Entities Early

DDS entities include DomainParticipants, Publishers, Subscribers, DataReaders and DataWriters. When creating a DataReader or DataWriter, the middleware allocates queues for the DataReader or DataWriter and sends discovery information to all the other applications announcing that the DataReader or DataWriter exists. The discovery process matches DataReaders and DataWriters in local applications to remote DataReaders and DataWriters in other applications. This process takes non-zero time to send over the network and shared memory. With typical QoS, DataReaders (even if they're reliable) only receive data after they've been discovered. If the user creates a DataWriter and immediately starts sending data, the DataWriter has probably not yet discovered the matching DataReader. Any data sent before it can be discovered by the DataReader will be dropped. In general, it's best to create these entities early so that discovery can be completed and memory can be allocated in advance.

Never Block a Callback

DDS listener classes provide methods that can be called by the middleware when certain events occur. These listeners can be installed on a DataReader, DataWriter, DomainParticipant, Publisher or Subscriber. Installing listeners on these entities allows the user to be notified about middleware events such as data becoming available, data being delayed or liveliness being lost.

Callbacks get called back from middleware threads. Middleware threads perform various actions such as receiving data from the socket buffer, taking it out, sending reliability metadata, checking that events are happening on time and maintaining liveliness.

If the user blocks these callbacks, it prevents the middleware from performing these actions – including receiving data. In Connext DDS, by default, multiple DataReaders may share the same thread for receiving data. This means if the user blocks the callback for one DataReader, it may cause a delay or loss of data for a different DataReader in the same application, which is extremely difficult to debug.

Consider this when writing a wrapper layer around DDS. Should listener callbacks be exposed to users of the wrapper layer? Developers who are writing code to use the wrapper layer are unlikely to know they should not block callbacks and may not be familiar with DDS. Developers who use the wrapper layer are likely to block callbacks and experience unexpected data loss.

Use WaitSets, Not Listeners

Unless a system requires extreme performance, using WaitSets can be a much safer option than listeners. WaitSets are a way to block one's own thread until data becomes available. Instead of having the middleware thread receive data and using that same thread to notify the application, the user can receive the notification in their own thread. From there, the developer can block or process the data however they want, without affecting the performance of Connext DDS. This makes WaitSets a much safer way to receive data.

There are some trade-offs. Many developers use listeners because they're the lowest-latency way to receive data. Using WaitSets does slightly increase latency due to the thread context switch. However, excluding cases that have an extreme performance requirement, WaitSets remain the safer option for receiving data and handling middleware events.

Network and QoS Best Practices

Application developers, system architects and platform developers may be responsible for following network and QoS best practices. These best practices involve how to tune the network or the QoS for the right amount of performance and scalability.

Use the Right Reliability

When starting out, most developers instinctually believe that all data needs to be highly reliable and highly available. In some cases this is true, but if the data has been modeled to be discrete updates about the state of an object, it does not have to be completely reliable. This can be beneficial in terms of overhead and memory usage. If DataWriters are configured to be only as reliable as necessary, the system gains performance and reduced memory usage.

Connext DDS has three types of reliability:

- **Strict reliability**: DataWriters keep a queue of data to re-send to DataReaders if the data is lost. DataWriters are not allowed to overwrite anything in their queue if an existing DataReader has not received it. If the DataWriter's queue fills, it will block and possibly return an error rather than overwriting data in the queue. This is configured by using the RELIABILITY QoS set to RELIABLE and the HISTORY QoS set to KEEP_ALL.
- **Non-strict reliability**: DataWriters keep a queue of data to re-send to DataReaders if the data is lost. DataWriters will reliably deliver the data that is in their queue, but are allowed to overwrite it if their queue is full – even if an existing DataReader has not received it. This is configured by using the RELIABILITY QoS set to RELIABLE and the HISTORY QoS set to KEEP_LAST.
- **Best-effort reliability**: DataWriters do not keep a queue of data, and do not use a reliability protocol to re-send lost data. This is configured by using the RELIABILITY QoS set to BEST_EFFORT.

Many developers default to using strict reliability. This is a reasonable choice for some types of data, if the system requirement states that every piece of data must be received by the DataReader. This is typically used for certain types of command data and event data. However, overuse can cause unnecessary CPU and bandwidth usage, when unnecessary data is sent to DataReaders.

Non-strict reliability is the best option for sending state data. Continuing the earlier example, each truck in the fleet keeps track of its maintenance information using the TruckMaintenance Topic. Each truck sends updates about whether it needs maintenance only if the state changes due to hitting a mileage limit. If the DataReader needs to know the current state of the truck – whether it needs maintenance or not – it should reliably receive this information. However, it should not receive the entire history of every time the truck needed maintenance – just the *current* maintenance state. This behavior can be achieved by setting the RELIABILITY QoS to RELIABLE, the HISTORY QoS to KEEP_LAST, and setting a HISTORY DEPTH to 1. This tells the DataWriter that it should reliably send just the current maintenance state to all interested DataReaders.

The last option is best effort reliability. When the middleware is configured for best effort reliability, it isn't providing any protocol to repair or resend any data that may have been lost on the network. Best effort data has no memory or protocol overhead. This option is typically used with rapid periodic sensor data. In our example, TruckGPS data is sent rapidly and periodically. It makes sense to send this best-effort to avoid the low but unnecessary overhead of the reliability protocol.

Use the Maximum Transport Sizes

In DDS terms, transport refers to the lower level communication mechanism, which may be UDPv4, UDPv6 or shared memory. Larger transport sizes can increase throughput of data. Different transports available can have different maximum data sizes. UDPv4 generally supports 64K, but if it's decreased by the sizes of various headers, it ends up being about 65507 bytes. Some platforms do not support the full 64K UDP packet size. TCP and shared memory can have maximum sizes larger than 64K.

If better throughput is needed, one of the first things to look at is changing transport sizes. If your system supports it, and you're using UDP, 65507 is a reasonable number to use as maximum transport size. If you're not using UDP, this size will be based on how much memory you want to use for transporting data, bandwidth limitations and limitations of the operating system.

Use Multicast for One-to-Many

In DDS terms, transport refers to the lower level communication mechanism, which may be UDPv4 or UDPv6. Connext DDS doesn't require use of multicast, but it is used by default for discovery. If multicast is available in a network, enabling it can significantly increase one-to-many performance. If a system includes one DataWriter sending high-throughput data to several DataReaders on different machines, enabling multicast can deliver a large throughput improvement. When using multicast, the DataWriter sends only a single multicast packet rather than one per DataReader. By enabling multicast, the DataWriter uses less CPU if it's writing rapid data. This results in increased throughput in one-to-many communications and reduced CPU usage by the DataWriter.

Multicast tradeoffs can generally be solved with QoS tuning or router configuration. The first tradeoff is that if the data DataWriter is reliable and the data is rapid, the user may need to tune the reliability data protocol to avoid acknowledgement (ACK) storms, which cause performance decreases. The second tradeoff is that some switches will block bursts of multicast traffic. To avoid this issue, make sure the network hardware is set up not to not block bursts of traffic.

Configure Switches for IGMP Snooping

If multicast is being used in a system, it's beneficial to configure switches for Internet Group Management Protocol (IGMP) snooping. Many switches treat multicast as broadcast traffic by default. This means that the DataWriter writes to a multicast address, but the switch broadcasts it to all the applications on the other side of the switch whether or not they are interested. Enabling IGMP snooping prevents switches from broadcasting to applications that are not interested. Instead, the switch filters data so that only the interested applications receive it. This can reduce congestion on the network and prevent applications from receiving (and having to filter) irrelevant data.

Even if multicast isn't being used for user data, it's used by to make the discovery process more efficient.

Tune the OS for Throughput

If a system has high throughput data, the user should look at tuning the operating system (OS) to get better throughput. Those who work with a real-time OS might be familiar with this, but desktop systems like Windows or Linux can also be tuned for better networking. With Windows, most people notice that the default settings are not optimized for high throughput UDP. To remedy this problem, in Windows registry, there's a setting that can be changed to yield better throughput. In Linux, there's the sysctl.conf control file that can be edited to achieve Embedded platforms will have settings specific to the platform.

Some of these best practices (along with technical details) can be found on RTI Community at community.rti.com/best-practices with more being added regularly. Questions and advice regarding best practices can be posted to the Community forum at community.rti.com/forum. For more information, visit www.rti.com or email info@rti.com.

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 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, California.

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