Data Centric Design for Networked Applications

In this article **Gordon A. Hunt**, principal engineer at Real-Time Innovations presents a data-oriented approach that enables seamless integration of different communication and data storage models in a real-time system

Today's embedded systems are becoming increasingly complex. Applications are becoming more distributed and individual systems (nodes) are becoming more heterogeneous. Additional complexity is added with real-time and dynamically-changing data requirements. Just to make the problem even more interesting, systems are required to enable seamless access to the data they contain through a variety of methods. Low-level messaging, publish/subscribe, data storage and SQL (Structured Query Language), and web service technologies are expected to be fully integrated, scalable and upgradeable in today's distributed applications.

By moving from a message-centric point-to-point solutions, which tend to be operating system specific and/or proprietary implementation oriented to standards-based data-centric technologies, we can develop systems that are inherently more robust, maintainable and upgradeable to meet changing customer and market requirements. Such a data-oriented approach decouples the system implementation in time, space and function, which significantly simplifies lifecycle development of the distributed system.

Recognising that it's the data that is critical in your system, by defining the data and its transient states you can completely define your system. You can then enable your application developers to use the development tools most familiar to them. Such an approach simplifies integration between nodes and addresses the issues of running on and connecting between heterogeneous real-time nodes and back-end Unix systems. For real-time access to data you can use standards-based publish/subscribe peer-to-peer technologies that facilitate high-speed deterministic connectivity, while your back-end system developers, which are more familiar with the enterprise space, can use SQL for their data processing needs and can access localised data or data on the real-time nodes. The following example will illustrate these concepts and ideas.

Example

The following diagram depicts a typical distributed system problem we are trying to solve, using this data-oriented approach. The goal in this example is to maintain the temperature in many buildings, using embedded controllers each hooked to a number of sensors. Each of these sensors and control processes are connected through a transport mechanism such as Ethernet, shared memory, or bus backplane technologies.

Basic protocols such as TCP-UDP/IP or higher-level protocols such as HTTP can be used to provide standardised communication paths between each of the nodes. To achieve data integrity and fail-over capabilities, multiple controllers and sensors can be deployed in each building. Additionally, depending on the size of the building, multiple controllers, each with appropriate backups could be distributed for the different zones. Controllers within a building need to collaborate and all data collected from the various sensors is stored real-time in web-accessible databases. With the inclusion of these distributed databases, we are providing a standards-based way for external applications to obtain, process and manipulate real-time sensor data without having to know the specifics of the real-time data infrastructure. The external access and monitoring applications can simply receive real-time updates from any sensor as well as issue commands to the various controllers via SQL, ensuring that optimal temperature is maintained. This simply stated example is surprisingly complex, containing many elements of real-time messaging, data integrity and failover capabilities, integration with databases, web services, as well as scalability and modularity concerns.

Data model

In order to simplify this example, we will only focus on the data the sensors send to their controller and how it can be distributed throughout the entire system. The first step in a data-centric approach is to carefully describe the data format in a standards-based way, either IDL or XML, and give it a "Topic" name. Topics are the element of the Data Distribution Service (DDS) middleware publish-subscribe standard (see sidebar) which identify the data objects and provide the basic connection between publishers and subscribers. Subscribers, in this case the Controllers, register Topics with the middleware they wish to receive. Publishers, the individual sensors in this example, register topics with the middleware they will send. If Topics do not match, communication will not take place.

Topics enable one to find specific information sources and sinks when architecting a loosely coupled system. A loosely coupled system is one in which you do not know a priori how many sensors or controllers there are going to be or where they all are. The controller can simply subscribe to "TempSensor", the Topic's name, and receive all the sensor updates for that building. Similarly, a sensor does not need to know if it is sending its data to one or multiple controllers.

Specification of the Topic's name is a key element in a data-centric approach to creating open real-time systems. One could name each sensor's topic based on its unique location in the building, "Floor12Room3Sensor14" for example, but the controller would then need to be configured every time a sensor is added or removed from the system. Topics (name and type) define the standard interface for the distributed system and should be chosen appropriately.

Data type

Specification of the Topic's data type is equally important as the Topic's name. For this example we are using Interface Definition Language (IDL) because it is an open standard and readily maps to XML and SQL semantics.

In the definition of the Topic's type, one or more data elements can be chosen to be a "Key". Keys provide scalability and the communication infrastructure can use the key to sort and order data from many sensors. In this example, without Keys, one would need to create individual Topics for each sensor. Topic names for these topics might be: Sensor_1, Sensor_2, and so on. Therefore, even though each Topic is comprised of the same data type, there would still be multiple Topics. With keys, there is only one topic, "TempSensor", used to report temperatures.

New sensors can be added without creating a new Topic. The publishing application would just need to set a new ID when it was ready to publish. An application can also have a situation where there are multiple publishers of the same Topic with the same Key defined. This enables the application to provide redundancy. Using our example, we can put two sensors in the same room, giving them the same Key value states so they are measuring the same piece of information. Managing the redundancy, should one or both sensors report to the controller, is accomplished though Quality-of-Service (QoS).

Data-centric QoS

Data-centric communication using DDS provides the ability to specify various parameters like the rate of publication, rate of subscription, how long the data is valid and many others. These QoS parameters allow system designers to construct a distributed application based on the requirements for, and availability of, each specific piece of data. A data-centric environment allows you to have a communication mechanism that is custom tailored to your distributed application's specific requirements yet remains a loosely coupled design and architecture.

The ability to set QoS on a per-entity basis is a significant capability provided by DDS. Being able to specify different QoS parameters for each individual Topic, Publisher or Subscriber, gives developers many options when designing their system. Through the combination of these parameters, a system architect can construct a distributed application to address an entire range of requirements, from simple communication patterns to complex data interactions.

The following briefly details how one might leverage a few of the QoS in DDS for this example.

➢ **Domain** – A Domain is the basic DDS construct used to bind individual publications and subscriptions together for communication. A distributed application can select to use single or multiple domains for its data-centric communications. In the example, different buildings map to different Domains. Domains isolate communication, promote scalability and segregate different classifications of data.

➢ **Partition** – The Partition QoS is a way to logically separate Topics within a Domain. The value is a string. If Subscriber sets this string, then it will only receive messages from Publishers that have set the same string. In the context of our example, Partitions can be used to group sensors on different floors. For example, we want to divide the building into different zones, where each zone is controlled by a dedicated controller, the sensor and controller could set the Partition to "Floor 1" and "Floor 1-6" respectively. Here, the controller will receive data from all sensors on floors 1 through 6. So, using Partitions make it easy to group the sensors that are 'hooked' to a controller. A controller can take over a different zone by changing or adding to its Partition list.

➢ **Ownership** – The Ownership QoS specifies whether or not multiple publishers can update the same data object and also how you achieve fault-tolerance using DDS.

Returning to our example, if we have multiple sensors in the same room and we only want to get data from the primary (as long as it is functioning) then the Ownership QoS policy is set to Exclusive, stating that only one sensor can update that keyed value. Setting the Ownership policy to Shared is stating that we can have multiple sensors in the same room all reporting the same piece of keyed data. In this case the controller would get all updates from all sensors and treat the values as the same measurement.

➢ **Durability** – The Durability QoS specifies whether past samples of data will be available to newly joining subscribers. Considering our example, if a controller were to reboot, rather than require all sensors to resend their data, or require the data to be sent at a periodic rate in case the system reboots, one simply gets the latest published value for every attached sensor. This effectively decouples the system in time and provides a high degree of data integrity.

➢ **History** – History specifies how many data samples will be stored for later delivery. Specifically, a rebooted controller may want the last five samples from its sensors, so that it can make sure that readings are consistent.

➢ **Reliablity** – Finally, the Reliability QoS may be set on a per Topic basis and informs the middleware that the Subscription should receive all data (no missed samples) from a Publication even over non-reliable transports. Generally, for periodic publications, Reliability doesn't need to be set since you can just get the updated value one sample period later. Although periodic sensor data doesn't need to be delivered reliably, synchronisation commands between Controllers in this example could be.

Integration with databases

The final element of our example system is the integration of real-time data and traditional relational databases. Since both these technologies are data-centric and complementary, they can be combined to enable a new class of applications. In particular, DDS can be used to enable a truly decentralised data structure for distributed database management system (DBMS),

while DBMS technology can be used to provide persistence for real-time DDS data.

Working with the example, each building can maintain the history of the various building sensors in a locally maintained database. The application that manages all buildings would have its own automatically maintained database of the specific data stores (tables) from each building that the application needed. Information is pushed to where it is needed, not senselessly replicated throughout the distributed system.

IDL data models can be automatically and cleanly mapped to SQL table schemas. For example, the Topic "TempSensor" becomes a table named "TempSensor" and the data contents, identified by the Key, become rows in the table.

Essentially, the database is simply another subscription to the sensors' update and automatically receives current data from all the distributed sensors. Changes to the database are pushed to entities that are interested in that particular topic/table name. Embedded applications don't need to know SQL or OBDC semantics, and the database applications don't need to know publish/subscribe semantics. This is a critical point when building large systems: get the data to where it needs to go in a format that is native to the developers.

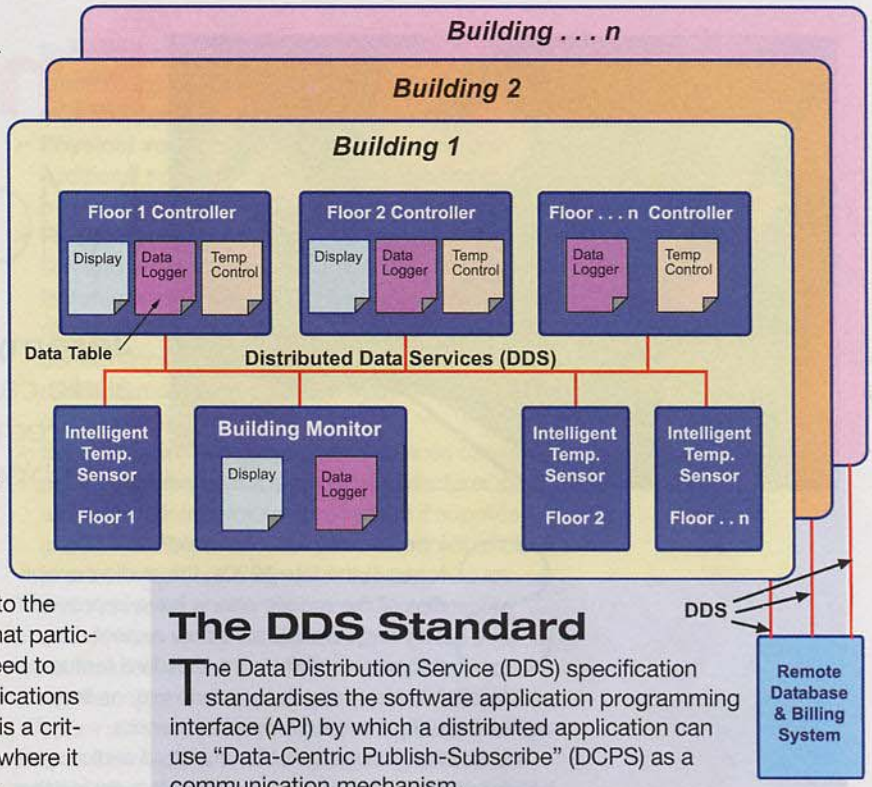
For true data integrity and scalability, databases should be distributed as well.

RTI's SkyBoard implements a distributed shared database, where fragments of the shared database are kept in the data caches of the hosts in the network on an as-needed basis. Thus, the database becomes a combination of the data stores distributed throughout the system. When a node updates a table by executing a SQL INSERT, UPDATE or DELETE statement on the data cache, the update is proactively pushed to other hosts that access this table via real-time publish-and-subscribe messaging, enabling real-time replication and synchronisation of any number of remote data stores.

Finally, once data is automatically entered and maintained in a DBMS, using standard tools, one can build a web application that accesses and manipulates the database data. Thus, the web application does not need to know how many buildings, sensors or controllers there are in the system. Nor does the web application need to know the middleware specifics that the temperature control system is using to distribute data. The application can just use SQL and ODBC to read and change all of the available real-time data in the system decoupling implementation specifics across the system.

Summary

By starting with the data model and designing the systems following a data-centric approach, we demonstrated building a system that seamlessly integrates a variety of different communication trends and data store trends (database and embedded local data types), while still achieving a high degree of data integrity.



The DDS Standard

The Data Distribution Service (DDS) specification standardises the software application programming interface (API) by which a distributed application can use "Data-Centric Publish-Subscribe" (DCPS) as a communication mechanism.

The DDS standard has three main goals:

1. To define a model for communication as pure data-centric exchanges, where applications publish (supply or stream) data, which is then available to remote applications that are interested in it.

2. To provide a mechanism of specifying the available resources and providing policies that allow the middleware to align the resources to the most critical requirements, giving system designers the ability to control Quality of Service (QoS) properties that affect predictability, overhead and resource utilisation.

3. To permit systems to scale to hundreds or thousands of publishers and subscribers in a robust manner.

Since DDS is implemented as an "infrastructure" solution, it can be added as the communication interface for any software application.

Advantages of DDS:

- Based on a simple "publish-subscribe" communication trend
- Flexible and adaptable architecture that supports "auto-discovery" of new or stale endpoint applications
- Low overhead – can be used with high-performance systems
- Deterministic data delivery
- Dynamically scalable, efficient use of transport bandwidth
- Supports one-to-one, one-to-many, many-to-one and many-to-many communications
- Large number of configuration parameters that give developers complete control of each message in the system.

DDS provides an infrastructure layer that enables many different types of applications to communicate with each other. The DDS specification is governed by the Object Management Group (OMG), which is the same organisation that governs the specifications for CORBA, UML and many other standards. A copy of the DDS specification can be obtained from the OMG website at www.omg.org