Applying DDS to Large Scale Mission Critical Distributed Systems
An experience report

Niels Kortstee
Technical lead
PrismTech
Email.niels.kortstee@prismtech.com
Agenda

- Common challenges in Large Scale Mission Critical Distributed Systems
- DDS use case: Grand Coulee Dam
  - Introduction/system overview
  - Challenges and solutions
- DDS use case: Coflight
  - Introduction/system overview
  - Challenges and solutions
- Conclusion
- Questions
Common Challenges
Common challenges

☐ Scalability
  ☐ Increasing number of communication endpoints
  ☐ Increasing data volume
  ☐ Increasing update frequency
  ☐ Limited resources / asymmetrical systems

☐ Complexity
  ☐ Large number of components with high number of dependencies

☐ Fault-tolerance
  ☐ Robust against failures (No single point of failure)
Common challenges

☐ Evolvability
  ☐ Ability to adapt to future changes and extensions
    ☐ Upgrades typically go step-by-step due to the complexity and size of these systems

☐ Security
  ☐ Protection against unauthorised access

☐ Safety
  ☐ Protection against unintended usage
Use case: Grand Coulee Dam
Introduction

- Gravity Dam on the Columbia River in the U.S. state of Washington
  - Built to produce hydroelectric power and provide irrigation
  - Reservoir supplies water for the irrigation of 671,000 acres (2,700 km²)
  - Largest electric power-producing facility in the US (6,809 MW)
  - One of the largest concrete structures in the world (11,975,520 cu yd/9,155,942 m³)

- DDS applied in Generic Data Acquisition and Control System (GDACS)
  - Large scale control system for Grand Coulee Dam (4 power plants, 33 generators)
  - Over 150 computing nodes
  - Network cables over a mile long
  - Tens of thousands of sensors
System overview

- **Sensors with PLC’s**
  - Measure physical quantities in the system (data acquisition)

- **Remote Terminal Units (RTU’s)**
  - Interface sensors with the distributed system using DDS (data communication)
    - Process and execute incoming operator commands
    - Make quantities available to interested parties
    - Monitor quantities and issue alarms based on alarm conditions

- **Master stations**
  - Aggregate and summarize quantities (data presentation)
  - Control equipment remotely (control)
System overview

- Data volume: over 500,000 quantities updated up to 100 times per second and potentially even more alarms
- Limited amount of memory and CPU on RTU’s
- Only unicast communication allowed in part of the system due to security regulations (Critical Infrastructure Protection standard)
- New types of sensors as well as RTU’s and master stations may be introduced in the future
- Recovery from software failure within a couple of seconds or faster if possible
- No possibility to switch off the system for upgrade or maintenance
Challenges & Solutions

- Data model
  - One generic Quantity Topic (evolvability, complexity)
    - Model struct with a union that holds all primitive types to allow any type of data
  - One instance per physical location per update frequency (scalability)
    - Model unbounded sequence of Quantity
    - Batch Quantities with the same update frequency in one sample

```c
typedef enum prim { INT, STR, BYTE } PRIM_TYPE;

typedef union value switch(PRIM_TYPE){
    case INT:
        long intValue;
    case STR:
        string strValue;
    case BYTE:
        octet octValue;
} qvalue;

struct quantity {
    long id;
    qvalue value;
};

struct quantities {
    string location;
    long freq;
    sequence<quantity> values;
};

#pragma keylist quantities location, freq
```
Challenges & Solutions

- Scalability: bandwidth usage
  - RTU → Master station
    - One unicast per DataReader
    - Bandwidth required: Data volume * #DataReader
  - One unicast per DataReader
    - Bandwidth required: Data volume

Bandwidth usage: not scalable
Challenges & Solutions

- **Scalability: Data Partitioning**
  - One partition per physical location

- **Scalability: Content-based re-batching & routing**
  - Re-batch and forward quantities based on expressed interest in partitions
Challenges & Solutions

- Scalability: Prevent router bottleneck
  - Introduce generic “Router” partition and forward any quantity there

- Example: RTU2 → All stations
  - RTU2 unicast to Router1
  - Router1 unicast to Router2
  - Router1 multicast to Station1-4
  - Router2 multicast to Station5-8

- Example: RTU6 → RTU3
  - RTU6 unicast to Router2
  - Router2 unicast to Router1
  - Router1 unicast to RTU3
Challenges & Solutions

- Fault-tolerance: Replicate components and arbitrate using DDS built-in capabilities
  - DeadlineQosPolicy
  - OwnershipQosPolicy
  - OwnershipStrengthQosPolicy

- Maintenance/upgrade is possible without switching off entire system
Use case: Coflight
Introduction

- The next generation air traffic control system for France, Italy and Switzerland
  - 5 centers in France
  - 4 centers in Italy
  - 2 centers in Switzerland

- DDS applied for Flight Data Processing
  - Performance
  - Reliability
  - Persistency
  - High availability
  - CORBA cohabitation
  - Long-term upgradability and scalability
System overview

- Up to 8,000 concurrently managed flight plans
  - Internal flight-plans: 350 KB
  - External flight-plans: 160 KB

- Up to 100 updates per second
  - Also persisted to disk for fault-tolerance

- Mixed CORBA/DDS and C++/Java environment

- Part of the system only has a 10 Mbit network connection

- System may suffer from temporary network disconnections
Challenges & Solutions

- Fault-tolerance: Reliability on publisher crash:
  - Published sample is multicasted to all subscribing nodes
  - Sample is received by one or more, but not all nodes
  - Publishing node crashes before resend could be performed
    - Result: Inconsistent view on data between subscribers

- Recovery from such an inconsistency required within 1 second
Challenges & Solutions

- Solution: Nodes that did receive data are made responsible for re-sending
  - Nodes maintain a timed buffer of received data from all nodes
  - Nodes detect crash of publishing node
  - Still alive nodes exchange last received sequence number from crashed node
  - If inconsistencies are detected, the node with the highest sequence number will perform re-send(s) to any node that misses data
Challenges & Solutions

- Fault-tolerance: publishing application needs to be able to synchronise on reception of a written sample by a (set of) DataReader(s)

- Solution: New boolean attribute “synchronous” introduced for the already existing ReliabilityQosPolicy

  - Synchronous DataWriter
    - Block until all “synchronous” DataReaders have acknowledged reception for maximally “max_blocking_time”

  - Synchronous DataReader
    - Acknowledge reception of every sample to DataWriter
Challenges & Solutions

- **Synchronous DataWriter:**
  - DataWriter blocks until synchronous DataReader acknowledgement has been received.

- **Non-synchronous DataWriter:**
  - Synchronous DataReader will acknowledge reception
  - DataWriter does not block for any acknowledgement
Challenges & Solutions

- Fault-tolerance: recover from temporary network disconnections after re-connection

- Split-brain syndrome
  - Two (sets of) nodes have a different perception on (part of) the historical data
  - Application-level decisions may be based on different input, so potentially dangerous to (re-)connect nodes

- Causes
  - Separately running (set of) nodes suddenly get connected to each other
  - Nodes get re-connected after being disconnected for a while
Challenges & Solutions

- Required behaviour is different for every use case and therefore needs to be configurable
  - No (re-)connection allowed – Prevent communication on network-level
    - Nodal configurable ‘ReconnectionAllowed’ option for networking
  - (Re-)connection allowed – Nodal configurable ‘merge policies’ for non-volatile data
    - Ignore  – Allow (re-)connection and take no corrective measures
    - Merge  – Allow (re-)connection and perform two-way merge of historical data
    - Replace – Allow (re-)connection and perform one-way merge of historical data
    - Delete  – Allow (re-)connection and delete all historical data
Challenges & Solutions

- Merge policy example
  - Deployment
    - Node N1 with DataWriter X
    - Node N2 with DataWriter Y
    - Allow reconnection
    - Perform 2-way merge after re-connect
  - Scenario
    - X on N1 writes sample X1
    - N1 and N2 disconnect
    - X on N1 writes sample X2
    - Y on N2 writes sample Y1
    - N1 and N2 reconnect
    - A two-way merge is performed
Challenges & Solutions

☐ Scalability: Part of the system only has a 10 Mbit connection (data volume > available bandwidth)
  ☐ Solution: Compress data that is sent over the network (using zlib)
    ☐ Serialized data is compressed before sending on the network
    ☐ Compressed data is decompressed after receiving from the network

☐ Security: Part of the system requires secure communication
  ☐ Solution: Encrypt data that is sent over the network using SSL

☐ Safety: The system should be protected against accidental application read and/or write access to data
  ☐ Solution: Allow configuration of read/write access configuration per partition-topic combination on nodal level
Conclusion
Conclusion

- DDS already provides most of the required capabilities to build large scale mission critical distributed systems
  - Data-centric approach
  - Extensive collection of Quality of Services
  - Standard API and network protocol

- Based on our experience with these systems, some specification extensions could be made:
  - Extended reliability
  - Compression
  - Encryption
  - Merge policies
  - Read/Write access

- Content-based routing facilities are necessary to satisfy scalability challenges for systems-of-systems and uni-cast only environments
Questions?