Practical Aspects of Using RTI Connext DDS in UGV

Josef Schröttle

Senior Systems Engineer

RUAG MRO Schweiz

RUAG Schweiz AG

Munich, May 22nd, 2019

Together ahead. RUAG

Short History of UGV at RUAG
Gecko, 2008 to 2011

- Vehicle Gecko, sequential hybrid with four hub motors
- Teleoperate/Control thru VHF radio
- Autonomy with preplanned/teach-in & follower
- Vehicle and Control Station in C++

RTI DDS in Control Station but not in vehicle yet!

Short History of UGV at RUAG
Eagle IV with VERO, 2012 to 2015

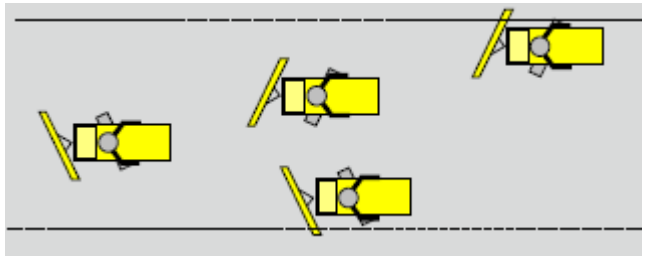
- Vehicle Robotics Kit VERO
- Autonomy: As Gecko with added collision avoidance using Radar

- Vehicle in C++ with DDS, control station rewrite in C#
- Concept for multiple control stations and multiple vehicles

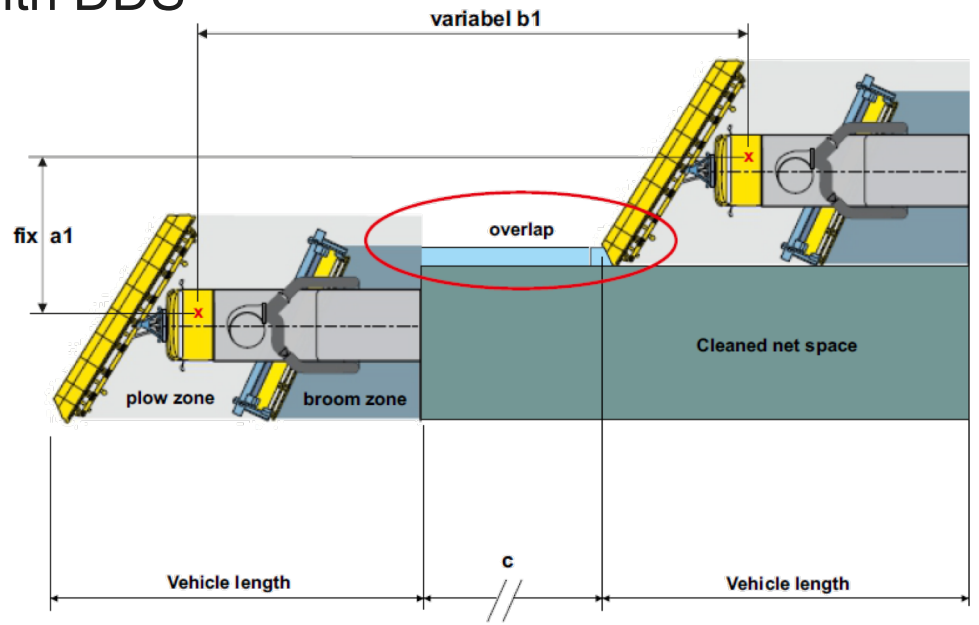
RTI DDS in Control Station and Vehicle!
Custom DDS-Router for multiple control stations & vehicles

Short History of UGV at RUAG
BASR, 2016 to 2018

- Boschung Automated Snow Removal
- Snow clearance on airports
- Vehicles in formation only



- Modular Architecture Lite/Full in C++ with DDS
- Vehicle to Vehicle Communication
- DDS-Routing thru 4G planned
- Trials in fall 2018 were promising

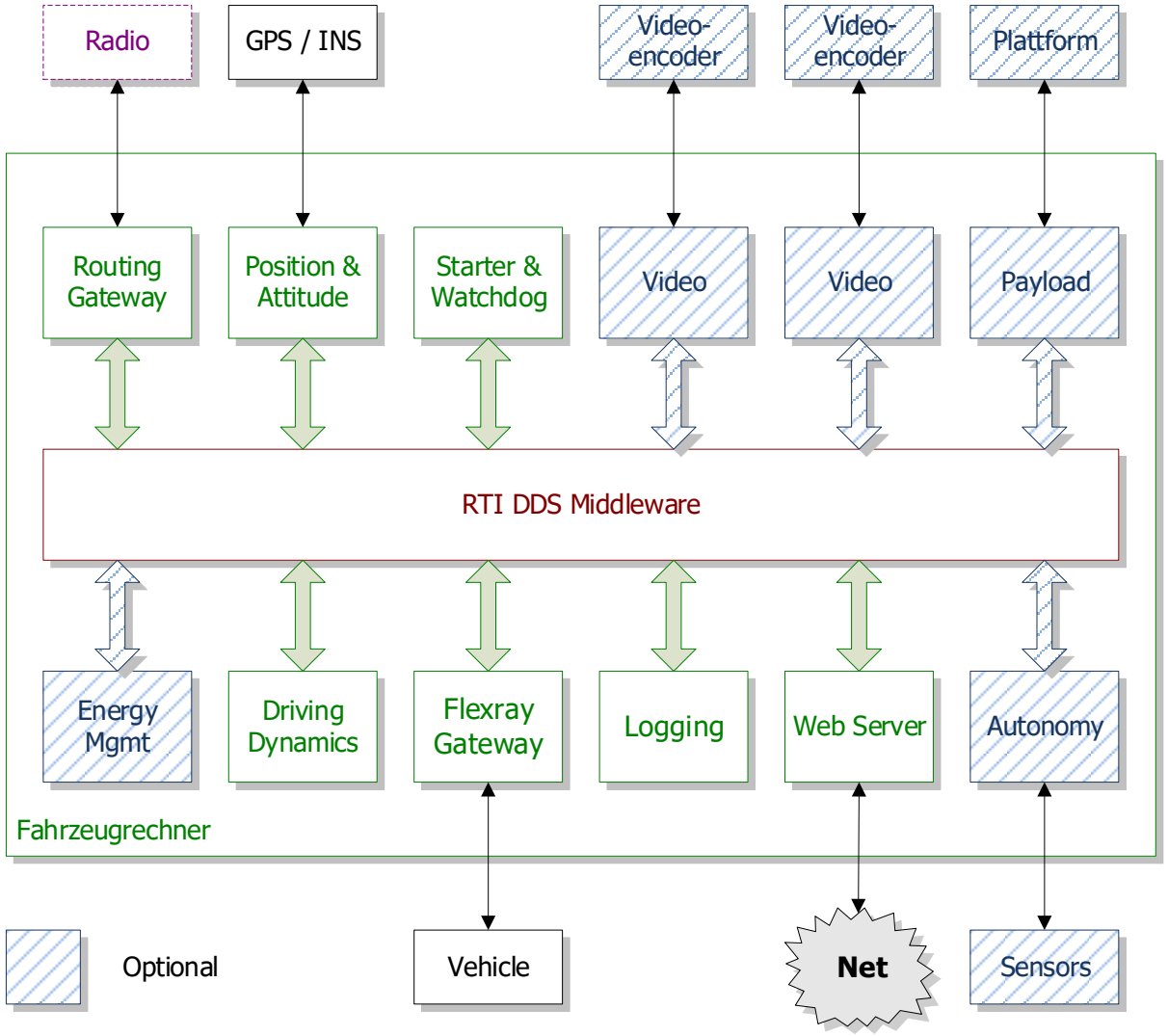


- End of robotics at RUAG due to company restructuring in 2019/20

Modular Architecture

- Almost everybody today talks about ‘system of systems’ or IoT
- This will be more and more important but is (in my opinion) mostly a communication and synchronization challenge
- Of course DDS will help you there
- But the ‘old’ software challenges still exist:
  - Modular Architecture
  - Maintainability
  - Testability
  - Scalability
- And DDS can help you there also
- We used DDS mainly to create a modular architecture for our control stations and vehicles

Modular Architecture in Vehicle



Modular Architecture in Vehicle and C2

- Implementation in the vehicle:
  - Every module is own Linux executable with DDS-Interface
  - Starter starts modules and monitors them (thru DDS lifeliness)

- Implementation in the control station is similar:
  - Every module is own Windows executable with DDS-Interface
  - Basic idea: Every window is an own executable
    e.g. dashboard, video in multiple instances, map

Modular Architecture in Vehicle and C2

- Advantages:
  - Modules can be optional and running in multiple instances (e.g. autonomy and video systems)
  - Implementation of modules either in central computer or separate hardware, changeable later without affecting other modules
  - The DDS middleware provides a fully documented layer of abstraction
  - Every module is independently testable on the DDS interface
  - Modules can be implemented in different programming languages
  - Many combinations possible for integration and testing e.g. Vehicle with Positioning, Logging and Web-Server
  - For testing the Routing Gateways can be eliminated and the DDS domains directly connected

- Challenges:
  - Process-Priorities must be configured correctly

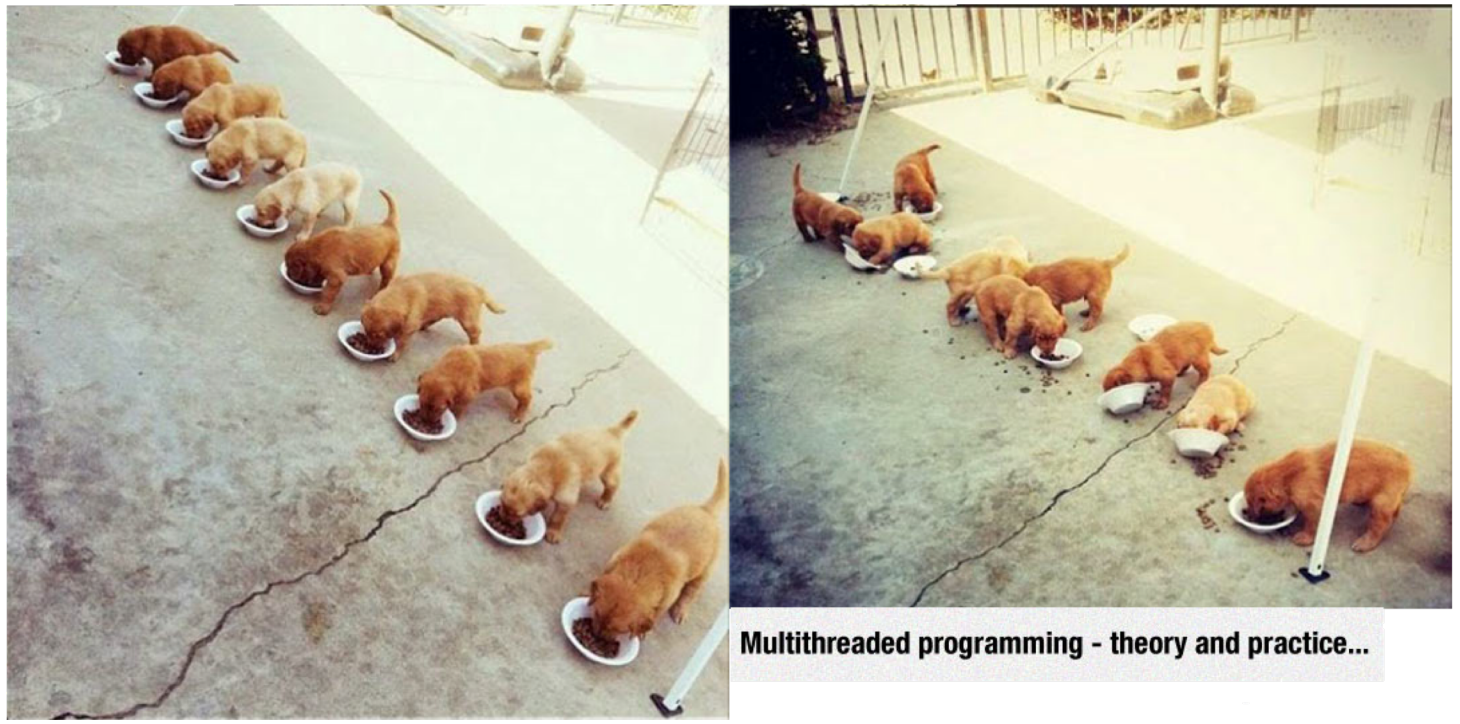
Adressing of Modules / Systems

- For modular systems all modules must conform at least to the following requirements:
  - Every module is addressable with an unique address
  - The source address is a ‘key’ in every topic
  - Every module implements a common interface with topics
    - Alive and status
    - Logistics data (e.g. software-version)
    - Error reporting & recovery
    - Logging

- Module addressing has evolved over time:
  - Gecko had only keys ‘AppID’ and ‘CEP’ (command&control only)
  - Eagle has ‘groupId’, ‘systemId’, ‘subsystemId’ and ‘’moduleId’
  - BASR has ‘vehicleId’, ‘typeId’ and ‘instanceId’ (inspired by GVA)

Now to practical aspects and examples

- Required knowledge for following samples:
  - C/C++
  - IDL files
  - RTI DDS QoS and profiles



Addressing of Modules / Systems
Source Address

- We defined a base structure for all topics and used inheritance to include it in all topic types
- The name was ‘BaseAV’ for ‘Actual Value’

```
// Basic Indentifier types
typedef unsigned long  VehicleId;
typedef unsigned short TypeId;
typedef unsigned short InstanceId;

// Address of an object
struct ObjectId
{
    VehicleId  vehicleId;
    TypeId     typeId;
    InstanceId instanceId;

}; //@top-level false
```

```
// Timestamp
struct Timestamp
{
    long long     seconds;
    unsigned long nanoseconds;

}; //@top-level false

// Base struct for all actual values
struct BaseAV
{
    ObjectId  sourceID; //@key
    Timestamp timeOfGeneration;

}; //@top-level false
```

Addressing of Modules / Systems
Source Address

- And it was used like this, e.g. in the logging topic

```
// System wide logging
struct LoggingAV : BaseAV
{
    Severity    severity;
    ShortString info;
    LongString  data;
    ErrorCode   errorCode;

}; //@Extensibility FINAL_EXTENSIBILITY
```

Addressing of Modules / Systems
Recipient Address = Directed Send

- Sometimes you need to send data specifically to a module
- This is contrary to the Pub / Sub idiom where the publisher should not know about the subscribers!

```
// Base struct for all nominal values
struct BaseNV
{
    ObjectId  sourceID; //@key
    ObjectId  recipientID; //@key
    Timestamp timeOfGeneration;

}; //@top-level false
```

```
// System wide error masking
struct ErrorMaskNV : BaseNV
{
    ErrorCode errorCode; //@key

}; //@Extensibility FINAL_EXTENSIBILITY
```

QoS Profiles

- Define QoS Profiles in XML and load the XML explicit during startup
- It is easily ‘adjustable’ during integration & debugging
- Define topic QoS only in profiles and create a ‘minimum’ working set
- Use inheritance to structure the profiles

```
<!-- Commands, keep the last value -->
<qos_profile name="BASR.Command" base_name="BASR.Base.KeepLastReliable"/>

<!-- Pulsed command, it disappears after the lifespan -->
<qos_profile name="BASR.Pulse" base_name="BASR.Base.StrictReliable.Volatile">
  <datawriter_qos>
    <lifespan>
      <duration>
        <sec>2</sec>
        <nanosec>DURATION_ZERO_NSEC</nanosec>
      </duration>
    </lifespan>
  </datawriter_qos>
</qos_profile>
```

Structure Topic Names

- Caution: All topic names are in a global namespace per domain (like global variables in a whole system)
- Make sure the names don‘t collide, like ‚Status‘ or ‚Mode‘
- Modules (namespace in IDL) don’t help here
- We used a structure in topic names, chars like ‘.’ or ‘/’ are possible

```
"BASR.Common.LoggingAV"
"BASR.Common.ErrorAV"
"BASR.Position.GlobalPositionAV"
"BASR.Navigation.SolutionAV"
"BASR.Navigation.AutonomousModeAV"
"BASR.MissionData.MissionNV"
"BASR.Gui.StatusAV"
"BASR.Safety.StatusAV"
```

Readable IDL Files

- It really helps to define topic type, name and QoS profile together
- The QoS profile is of the writer, the reader can differ (but mostly is same)

```
// System wide logging
struct LoggingAV : BaseAV
{
    Severity    severity;
    ShortString info;
    LongString  data;
    ErrorCode   errorCode;

}; //@Extensibility FINAL_EXTENSIBILITY

// Topic name and QoS profile
const string TOPIC_LOGGING_AV   = "BASR.Common.LoggingAV";
const string PROFILE_LOGGING_AV = "BASR.Logging";
```

Be Aware of Traffic
QoS Minimum Separation

- Sometimes a publisher has data with a high frequency
  e.g. INU attitude data with 50 Hz or more
- And has multiple subscribers like Navigation and GUI
- If the GUI now just subscribes it will ‘wake up’ way to often
  (a typical GUI update rate is 10 Hz for a ‘smooth’ display)

- The solution is QoS 'Minimum Separation’
  – to reduce CPU load and to avoid 'sample lost'
  – to reduce traffic

- The INU writer publishes with Best-Effort profile `"BASR.PeriodicData"`
- The Navigation reader uses the same profile to get max. speed
- The GUI reader uses the profile `"BASR.PeriodicData.10Hz"`
  which has a minimum separation of 100ms (= 10 Hz)

Be Aware of Traffic
Content Filter

- Often applications need topic data from a specific source only
- E.g. our vehicles listen only to their ‘assigned’ control station
- But all control stations publish on the same topic (keyed with sourceID)
- ‘Normally’ applications wake up on data, see if it is for them and throw the sample away if it is not for them (in reader callback code)

- But the discarded sample has already generated traffic and CPU load!

- The solution is to set a ‘Content Filter’ on the sourceID in the reader
- The filtering will then be done in the writer separately for every reader!

- Note: Set the content filter before creation of the reader
  (otherwise the 'historical' samples won't be filtered)

Writer and Reader Lifecycle

- Create all writer & readers at application startup
  - do not create writers 'on demand’
  - do not destroy writers
  - You can create readers ‘on demand’ but try to avoid it
- Create writers before readers
  - most likely writer topics will be updated on response to reader data
  - You cannot (and should not) create writers in reader listeners
- When specifying a listener in create_datareader() be prepared to handle data of the reader when the reader is created
  (reader listeners can be called before create_datareader() returns!)

- We used a two-stage approach (init/run) where all writers & readers were created in ‘init’ and all listeners / waitsets started in ‘run’
- Generally prefer Waitsets over Listeners!

Application Error Reporting
Motivation

- For a real-world system you need to have a ‘standardized’ way to report errors and react to them
- In our system the ‘error topic’ has evolved over time:
  - Gecko: Topic with 64-bit bitfield in an ‘unsigned long long’
  - VERO: Topic with sequence<unsigned short>

- Problem of this approach: Setting / Clearing an error is a Read-Modify-Write operation on the writer side and therefore needs a global variable and locking!

- Solution:
  - BASR: Keyed topic with error number as key
  - Use ‘Write’ to set an error, ‘Dispose’ to clear an error
  - No locking required

Application Error Reporting
Example

```
module basr
{
    module common
    {
        // Suggested action to fix an error. Will be sent with the error.
        enum FixErrorMethod
        {
            FIX_GOOD = 0,           // No error, nothing to do
            FIX_RESET_MODULE,       // Reset module thru Reset() function
            FIX_RESTART_MODULE,     // Restart module thru unload/load
            FIX_RESTART_SYSTEM,     // Restart whole system ('warm start')
            FIX_POWER_CYCLE,        // Cold start system
            FIX_MAINTENANCE         // Call maintenance
        };

        // Basic Error Code
        typedef unsigned long ErrorCode;

        // Common severity using QtMsgType
        typedef unsigned long Severity;
```

Application Error Reporting
Example, cont.

```
        // System wide error reporting
        struct ErrorAV : BaseAV
        {
            ErrorCode       code; //@key
            FixErrorMethod fixMethod;
            Severity        severity;

            // There is intentionally no string here!
            // The GUI needs to have error messages in all supported languages
            // Detailed error infos should be saved in the log

        }; //@Extensibility FINAL_EXTENSIBILITY

        // Topic name and QoS
        const string TOPIC_ERROR_AV   = "BASR.Common.ErrorAV";
        const string PROFILE_ERROR_AV = "BASR.Data";

    }; // end of namespace 'common‘

}; // end of namespace 'basr'
```

Summary, Lessions Learned
'Best Practices' or 'Do's'

- Use QoS profiles and a minimum ‘understandable’ set of them
- Structure Topic Names
- Create ‘readable’ IDL files containing type, name and writer QoS profile
- Use modules (namespaces) in IDL files to structure types
- Use content filter and minimum separation to reduce traffic
- Create writers before readers and never destroy them
- Define an error reporting scheme based on keys

Challenges
when using DDS in Real-World-Applications

- Time Sync of all nodes is required
  - The good news are that encryption is then also possible
  - Typical symptom: Data flows only in one direction
  - We used a local NTP server, GPS time or both
  - Challenge: If the time is really ‚off‘, e.g. 1.1.1970, then a ‚jump‘ is needed and we need to wait for time-sync before starting DDS apps

- Discovery
  - Discovery is not 'realtime'
  - Default Discovery uses Multicast
    - You do not want multicast via radio links
    - WLAN falls back to lowest available speed on multicast frames
    - Switches turn to hubs when IGMP is not enabled
  - Static Discovery increases application startup time

Challenges
when using DDS in Real-World-Applications, cont

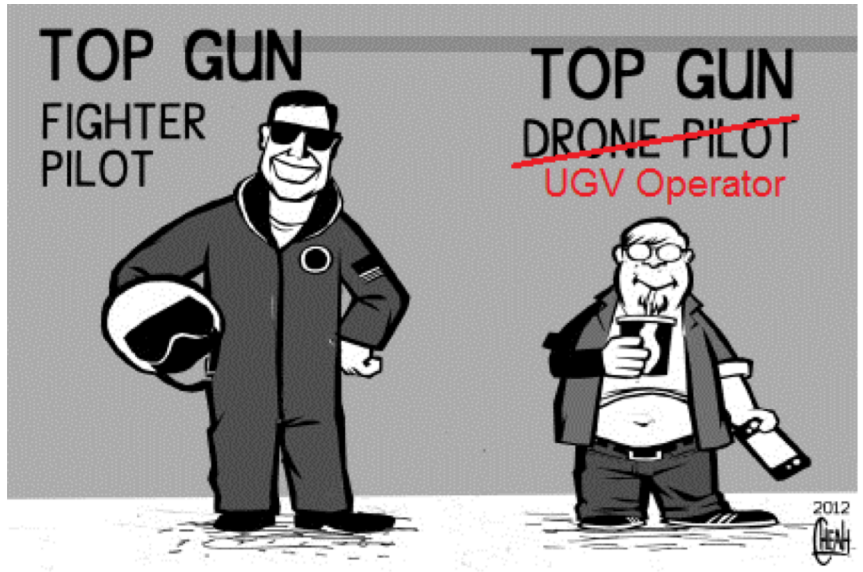
- MaxMTU
  - Yocto Linux typically has MaxMTU < 1500 and no fragments
  - Discovery typecode size exceeds MaxMTU in real-world scenarios
  - Topic data size can exceed MaxMTU

- RTPS2 over narrow radio links
  - Verbose, minimum header size > 200 bytes
  - Compression negligible
  - In our routing gateway the sender strips the RTPS2 header and the receiver attaches it again (but this has some drawbacks and is a lot more complicated than it sounds)

- Be aware that ‚normal‘ writers send UDP packets to all discovered readers separately
  - This scales quickly, e.g. 10 writers to 10 readers = 100 UDP packets
  - A multicast writer would reduce this to 10 packets

Thank you for your attention!

I’m looking forward to answer your questions



Together ahead. RUAG