Evaluating the Performance of Pub/Sub Platforms for Tactical Information Management

Dr. Douglas C. Schmidt
d.schmidt@vanderbilt.edu

Jeff Parsons
j.parsons@vanderbilt.edu

Ming Xiong
xiongm@isis.vanderbilt.edu

James Edmondson
jedmondson@gmail.com

Hieu Nguyen
hieu.t.nguyen@vanderbilt.edu

Olabode Ajiboye
olabode.ajiboye@vanderbilt.edu

July 11, 2006

Research Sponsored by AFRL/IF, NSF, & Vanderbilt University
Demands on Tactical Information Systems

Key *problem space* challenges

- Large-scale, network-centric, dynamic, systems of systems
- Simultaneous QoS demands with insufficient resources
  - e.g., wireless with intermittent connectivity
- Highly diverse & complex problem domains

Key *solution space* challenges

- Enormous accidental & inherent complexities
- Continuous technology evolution refresh, & change
- Highly heterogeneous platform, language, & tool environments
Promising Approach: The OMG Data Distribution Service (DDS)

Provides flexibility, power & modular structure by decoupling:

- **Location** – anonymous pub/sub
- **Redundancy** – any number of readers & writers
- **Time** – async, disconnected, time-sensitive, scalable, & reliable data distribution at multiple layers
- **Platform** – same as CORBA middleware
Overview of the Data Distribution Service (DDS)

- A highly efficient OMG pub/sub standard
  - Fewer layers, less overhead
  - RTPS over UDP will recognize QoS

- RT Info to Cockpit & Track Processing
- Tactical Network & RTOS
Overview of the Data Distribution Service (DDS)

• A highly efficient OMG pub/sub standard
  • Fewer layers, less overhead
  • RTPS over UDP will recognize QoS
• DDS provides meta-events for detecting dynamic changes
Overview of the Data Distribution Service (DDS)

• A highly efficient OMG pub/sub standard
  • Fewer layers, less overhead
  • RTPS over UDP will recognize QoS
• DDS provides meta-events for detecting dynamic changes
• DDS provides policies for specifying many QoS requirements of tactical information management systems, e.g.,
  • Establish contracts that precisely specify a wide variety of QoS policies at multiple system layers
Overview of DDS Implementation Architectures

• Decentralized Architecture
  – embedded threads to handle communication, reliability, QoS etc
Overview of DDS Implementation Architectures

- **Decentralized Architecture**
  - embedded threads to handle communication, reliability, QoS etc

- **Federated Architecture**
  - a separate daemon process to handle communication, reliability, QoS, etc.
Overview of DDS Implementation Architectures

- **Decentralized Architecture**
  - embedded threads to handle communication, reliability, QoS etc

- **Federated Architecture**
  - a separate daemon process to handle communication, reliability, QoS, etc.

- **Centralized Architecture**
  - one single daemon process for domain
DDS1 (Decentralized Architecture)

**Pros:** Self-contained communication end-points, needs no extra daemons

**Cons:** User process more complex, e.g., must handle config details (efficient discovery, multicast)
DDS2 (Federated Architecture)

Pros: Less complexity in user process & potentially more scalable to large # of subscribers
Cons: Additional configuration/failure point; overhead of inter-process communication
DDS3 (Centralized Architecture)

Pros: Easy daemon setup
Cons: Single point of failure; scalability problems
## Architectural Features Comparison Table

<table>
<thead>
<tr>
<th>QoS</th>
<th>Description</th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notification Mechanism</td>
<td>Blocking or Non-blocking data receiving</td>
<td>Listener-Based/Wait-Based</td>
<td>Listener-Based/Wait-Based</td>
<td>Listener-Based</td>
</tr>
<tr>
<td>Transport</td>
<td>Controls whether to use network multicast/broadcast/unicast addresses when sending data samples to DataSenders</td>
<td>Unicast/Multicast</td>
<td>Broadcast/Multicast</td>
<td>Unicast + transport framework</td>
</tr>
<tr>
<td>Higher-level DDS Protocol</td>
<td>On-the-wire communication model</td>
<td>RTPS Like protocol</td>
<td>RTPS Like protocol</td>
<td>N/A</td>
</tr>
<tr>
<td>Lower-level Transport</td>
<td>Underlying communication transport</td>
<td>Shared Memory/UDPv4</td>
<td>Shared Memory/UDPv4</td>
<td>Simple TCP/Simple UDP</td>
</tr>
</tbody>
</table>
# QoS Policies Comparison Table (partial)

<table>
<thead>
<tr>
<th>QoS</th>
<th>Description</th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DURABILITY</strong></td>
<td>Controls how long published samples are stored by the middleware for late-joining data readers</td>
<td>VOLATILE TRANSIENT-LOCAL</td>
<td>VOLATILE TRANSIENT-LOCAL PERSISTENT</td>
<td>VOLATILE</td>
</tr>
<tr>
<td><strong>HISTORY</strong></td>
<td>Sets number of samples that DDS will store locally for data writers &amp; data readers</td>
<td>KEEP_LAST KEEP_ALL</td>
<td>KEEP_LAST KEEP_ALL</td>
<td>KEEP_LAST KEEP_ALL</td>
</tr>
<tr>
<td><strong>RELIABILITY</strong></td>
<td>Whether data published by a data writer will be reliably delivered by DDS to matching data readers</td>
<td>BEST_EFFORT RELIABLE</td>
<td>BEST_EFFORT RELIABLE</td>
<td>BEST_EFFORT(UDP) RELIABLE(TCP)</td>
</tr>
<tr>
<td><strong>RESOURCE_LIMITS</strong></td>
<td>Controls memory resources that DDS allocates &amp; uses for data writer or data reader</td>
<td>initial_instance(extension) initial_samples(extension) max_instances max_samples max_samples_per_instance</td>
<td>max_instances max_samples max_samples_per_instance</td>
<td>max_instances max_samples max_samples_per_instance</td>
</tr>
</tbody>
</table>
Evaluation Focus

• Compare performance of C++ implementations of DDS to:
  • Other pub/sub middleware
    • CORBA Notification Service
    • SOAP
    • Java Messaging Service
  • DDS? JMS? SOAP? Notification Service?
Evaluation Focus

• Compare performance of C++ implementations of DDS to:
  • Other pub/sub middleware
    • CORBA Notification Service
    • SOAP
    • Java Messaging Service
  • Each other
Evaluation Focus

- Compare performance of C++ implementations of DDS to:
  - Other pub/sub middleware
    - CORBA Notification Service
    - SOAP
    - Java Messaging Service
  - Each other

- Compare DDS portability & configuration details
Evaluation Focus

• Compare performance of C++ implementations of DDS to:
  • Other pub/sub middleware
    • CORBA Notification Service
    • SOAP
    • Java Messaging Service
  • Each other

• Compare DDS portability & configuration details

• Compare performance of subscriber notification mechanisms
  • Listener vs. wait-set
Overview of ISISlab Testbed

Platform configuration for experiments

- **OS**: Linux version 2.6.14-1.1637_FC4smp
- **Compiler**: g++ (GCC) 3.2.3 20030502
- **CPU**: Intel(R) Xeon(TM) CPU 2.80GHz w/ 1GB ram
- **DDS**: Latest C++ versions from 3 vendors

wiki.isis.vanderbilt.edu/support/isislab.htm has more information on ISISlab
Benchmarking Challenges

• Challenge – Measuring latency & throughput accurately without depending on synchronized clocks

• Solution
  – Latency – Add ack message, use publisher clock to time round trip
  – Throughput – Remove sample when read, use subscriber clock only
Benchmarking Challenges

• Challenge – Measuring latency & throughput accurately without depending on synchronized clocks
• Solution
  – Latency – Add ack message, use publisher clock to time round trip
  – Throughput – Remove sample when read, use subscriber clock only

• Challenge – Managing many tests, payload sizes, nodes, executables
• Solution – Automate tests with scripts & config files
Benchmarking Challenges

• Challenge – Measuring latency & throughput accurately without depending on synchronized clocks

• Solution
  – Latency – Add ack message, use publisher clock to time round trip
  – Throughput – Remove sample when read, use subscriber clock only

• Challenge – Managing many tests, payload sizes, nodes, executables

• Solution – Automate tests with scripts & config files

• Challenge – Calculating with an exact # of samples in spite of packet loss

• Solution – Have publisher ‘oversend’, use counter on subscriber
Benchmarking Challenges

• Challenge – Measuring latency & throughput accurately without depending on synchronized clocks
  • Solution
    – Latency – Add ack message, use publisher clock to time round trip
    – Throughput – Remove sample when read, use subscriber clock only
• Challenge – Managing many tests, payload sizes, nodes, executables
  • Solution – Automate tests with scripts & config files
• Challenge – Calculating with an exact # of samples in spite of packet loss
  • Solution – Have publisher ‘oversend’, use counter on subscriber
• Challenge – Ensuring benchmarks are made over ‘steady state’
  • Solution – Send ‘primer’ samples before ‘stats’ samples in each run
  – Bounds on # of primer & stats samples
    • Lower bound – further increase doesn’t change results
    • Upper bound – run of all payload sizes takes too long to finish
const short MAX_MSG_LENGTH = 16384;

struct PubMessage {
    long seqnum;
    sequence<octet, MAX_MSG_LENGTH> data;
};

struct AckMessage {
    long seqnum;
};

DDS Latency And Jitter

Latency & jitter on same node

Seq. lengths in powers of 2 to upper bound

Ack message of 4 bytes

Tested seq. of bytes

Latency & jitter on different nodes
1-to-1 Single Node Latency

The graph illustrates the round trip latency (in us) for different data sizes (in bytes). The data is categorized into three groups: DDS1, DDS2, and DDS3, each represented by a distinct line on the graph. The x-axis represents the data sizes, ranging from 4 to 16384 bytes, while the y-axis shows the round trip latency ranging from 0 to 350 us.

- DDS1: Represented by blue diamonds, showing consistent latency across different data sizes.
- DDS2: Represented by pink squares, also showing consistent latency.
- DDS3: Represented by green triangles, showing a slight increase in latency as the data size increases.

The graph provides a visual comparison of how each DDS performs under varying data sizes, indicating the efficiency and latency characteristics for each category.
1-to-1 Single Node Latency

DDS3 is slower – UDP loopback instead of shared memory

DDS1 and DDS2 perform better due to shared memory transports
1-to-1 Single Node Jitter

Data Sizes (in bytes)

Round Trip Jitter (in us)

DDS1

DDS2

DDS3
1-to-1 Single Node Jitter

For larger payloads, DDS3’s lack of shared memory takes a toll

Even with shared memory vs loopback, jitter is well-paced for smaller payloads
1-to-1 Multiple Node Latency

Round Trip Latency (in us)

Data Sizes (in bytes)

- DDS1
- DDS2
- DDS3
1-to-1 Multiple Node Latency

DDS2’s federated architecture incurs extra context switching, synchronization and data copying.
1-to-1 Multiple Node Jitter

Data Sizes (in bytes)

Round Trip Jitter (in us)

- DDS1
- DDS2
- DDS3
The extra overhead of DDS2's federated architecture is evident in the jitter as well.

Federated arch. is designed for scalability, low CPU usage, not low latency.
Scaling Up DDS Subscribers

- The past 8 slides showed latency/jitter results for 1-to-1 tests
- We now show throughput results for 1-to-N tests

Publisher oversends to ensure sufficient received samples

4, 8, & 12 subscribers each on different blades

Byte sequences

Seq. lengths in powers of 2 (4 – 16384)

100 primer samples

10,000 stats samples

All following graphs plot median + “box-n-whiskers” (50%ile-min-max)
Scaling Up Subscribers – DDS1 Unicast

Performance increases linearly for smaller payloads

Performance levels off for larger payloads

- subscriber uses listener
- no daemon (app spawns thread)
- KEEP_LAST (depth = 1)
Scaling Up Subscribers – DDS1 Multicast

Performance increases more irregularly with # of subscribers

- subscriber uses listener
- no daemon (library per node)
- KEEP_LAST (depth = 1)

Performance levels off less than for unicast

4 Subscribers 8 Subscribers 12 Subscribers
Throughput greater for multicast over almost all payloads

Performance levels off less for multicast

- subscriber uses listener
- no daemon (app spawns thread)
- KEEP_LAST (depth = 1)
Scaling Up Subscribers – DDS1 1 to 8

Greater difference than for 4 subscribers

Performance levels off less for multicast

- subscriber uses listener
- no daemon (app spawns thread)
- KEEP_LAST (depth = 1)
Scaling Up Subscribers – DDS1 1 to 12

Greater difference than for 4 or 8 subscribers

Difference most pronounced with large payloads

- subscriber uses listener
- no daemon (app spawns thread)
- KEEP_LAST (depth = 1)
Scaling Up Subscribers – DDS2 Broadcast

Less throughput reduction with subscriber scaling than with DDS1

Performance continues to increase for larger payloads

- subscriber uses listener
- daemon per network interface
- KEEP_LAST (depth = 1)

4 Subscribers  8 Subscribers  12 Subscribers
Scaling Up Subscribers – DDS2 Multicast

- Lines are slightly closer than for DDS2 broadcast

- subscriber uses listener
- daemon per network interface
- KEEP_LAST (depth = 1)

4 Subscribers
8 Subscribers
12 Subscribers
Scaling Up Subscribers – DDS2 1 to 4

Multicast performs better for all payload sizes

- subscriber uses listener
- daemon per network interface
- KEEP_LAST (depth = 1)
Scaling Up Subscribers – DDS2 1 to 8

Performance gap slightly less than with 4 subscribers

- subscriber uses listener
- daemon per network interface
- KEEP_LAST (depth = 1)
Scaling Up Subscribers – DDS2 1 to 12

Broadcast/multicast difference greatest for 12 subscribers

- subscriber uses listener
- daemon per network interface
- KEEP_LAST (depth = 1)
Scaling Up Subscribers – DDS3 Unicast

Throughput decreases dramatically with 8 subscribers, less with 12

Performance levels off for larger payloads

- subscriber uses listener
- centralized daemon
- KEEP_ALL

4 Subscribers  8 Subscribers  12 Subscribers
Impl Comparison: 4 Subscribers Multicast

- DDS1 faster for all but the very smallest & largest payloads
- Multicast not supported by DDS3

- subscriber uses listener
- KEEP_LAST (depth = 1)
Slightly more performance difference for 8 subscribers

Multicast not supported by DDS3

- subscriber uses listener
- KEEP_LAST (depth = 1)
Impl Comparison: 12 Subscribers Multicast

Slightly less separation in performance with 12 subscribers

Multicast not supported by DDS3

- subscriber uses listener
- KEEP_LAST (depth = 1)
Impl Comparison: 4 Subscribers Unicast

DDS1 significantly faster except for largest payloads

Unicast not supported by DDS2

- subscriber uses listener
- KEEP_ALL
Impl Comparison: 8 Subscribers Unicast

- Performance differences slightly less than with 4 subscribers
- Unicast not supported by DDS2

- subscriber uses listener
- KEEP_ALL
Impl Comparison: 12 Subscribers Unicast

- Performance differences slightly less than with 8 subscribers
- Unicast not supported by DDS2
- Subscriber uses listener
- KEEP_ALL

Graph showing KB/sec vs. Bytes with DDS1 and DDS3.
Overview of DDS Listener vs. Waitset

Subscriber Application

- Data Reader
- Listener
- DDS
- on_data_available()

Key characteristics:
- No application blocking
- DDS thread executes application code

Subscriber Application

- Data Reader
- Waitset
- Condition
- Condition
- Condition
- DDS
- take_w_condition()

Key characteristics:
- Application blocking
- Application has full control over priority, etc.
Comparing Listener vs Waitset Throughput

Publisher oversends to ensure sufficient received samples

4 subscribers on different blades

Blade 0

Byte sequences

Seq. lengths in powers of 2 (4 – 16384)

100 primer samples 10,000 stats samples

Blade 4

Blade 3

Blade 2

Blade 1
Impl Comparison: Listener vs. Waitset

DDS1 – listener outperforms waitset & DDS2 (except for large payloads)

No consistent difference between DDS2 listener & waitset

- multicast
- 4 subscribers
- KEEP_LAST (depth = 1)
DDS Application Challenges

• Scaling up number of subscribers
• Data type registration race condition (DDS3)
• Setting proprietary ‘participant index’ QoS (DDS1)
DDS Application Challenges

- Scaling up number of subscribers
  - Data type registration race condition (DDS3)
- Setting proprietary ‘participant index’ QoS (DDS1)
- Getting a sufficient transport buffer size
DDS Application Challenges

- Scaling up number of subscribers
  - Data type registration race condition (DDS3)
  - Setting proprietary ‘participant index’ QoS (DDS1)
- Getting a sufficient transport buffer size
- QoS policy interaction
  - HISTORY vs RESOURCE LIMITS
    - KEEP_ALL => DEPTH = <INFINITE>
    - no compatibility check with RESOURCE LIMITS
    - KEEP_LAST => DEPTH = n
      - can be incompatible with RESOURCE LIMITS value
## Portability Challenges

<table>
<thead>
<tr>
<th></th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainParticipant Factory</td>
<td>compliant</td>
<td>compliant</td>
<td>proprietary function</td>
</tr>
<tr>
<td>Register Data Types</td>
<td>static method</td>
<td>member method</td>
<td>member method</td>
</tr>
<tr>
<td>Spec Operations</td>
<td>extra argument (newer spec)</td>
<td>compliant</td>
<td>compliant</td>
</tr>
<tr>
<td>Key Declaration</td>
<td>//@key</td>
<td>single #pragma</td>
<td>pair of #pragma</td>
</tr>
<tr>
<td>Required App. IDs</td>
<td>publisher &amp; subscriber</td>
<td>none</td>
<td>publisher</td>
</tr>
<tr>
<td>Required App. Transport Config</td>
<td>code-based</td>
<td>none</td>
<td>file-based or code-based</td>
</tr>
</tbody>
</table>
## Portability Challenges

<table>
<thead>
<tr>
<th></th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainParticipant Factory</td>
<td>compliant</td>
<td>compliant</td>
<td>proprietary function</td>
</tr>
<tr>
<td>Register Data Types</td>
<td>member function</td>
<td>member method</td>
<td>member method</td>
</tr>
<tr>
<td></td>
<td><code>DomainParticipantFactory::get_instance();</code></td>
<td><code>TheParticipantFactoryWithArgs(argc, argv);</code></td>
<td><code>TheParticipantFactoryWithArgs(argc, argv);</code></td>
</tr>
<tr>
<td>Spec Operations</td>
<td>extra argument (newer spec)</td>
<td>compliant</td>
<td>compliant</td>
</tr>
<tr>
<td>Key Declaration</td>
<td>single pair of #pragma</td>
<td>single pair of #pragma</td>
<td>single pair of #pragma</td>
</tr>
<tr>
<td>Required App. IDs</td>
<td>publisher &amp; subscriber</td>
<td>none</td>
<td>publisher</td>
</tr>
<tr>
<td>Required App. Transport Config</td>
<td>code-based</td>
<td>none</td>
<td>file-based or code-based</td>
</tr>
</tbody>
</table>
# Portability Challenges

<table>
<thead>
<tr>
<th></th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DomainParticipant Factory</strong></td>
<td>compliant</td>
<td>compliant</td>
<td>proprietary function</td>
</tr>
<tr>
<td><strong>Register Data Types</strong></td>
<td>static method</td>
<td>member method</td>
<td>member method</td>
</tr>
<tr>
<td></td>
<td>extra argument</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>DataType::register_type(participant, name);</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Key Declaration</strong></td>
<td>//@key</td>
<td>single</td>
<td>pair of pragma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>DataType identifier;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>identifier.register_type(participant, name);</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Required App. IDs</strong></td>
<td>subscriber</td>
<td>none</td>
<td>publisher</td>
</tr>
<tr>
<td><strong>Required App. Transport Config</strong></td>
<td>code-based</td>
<td>none</td>
<td>file-based or code-based</td>
</tr>
</tbody>
</table>
### Portability Challenges

<table>
<thead>
<tr>
<th></th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DomainParticipant Factory</strong></td>
<td>compliant</td>
<td>compliant</td>
<td>proprietary function</td>
</tr>
<tr>
<td><strong>Register Data Types</strong></td>
<td>static method</td>
<td>member method</td>
<td>member method</td>
</tr>
<tr>
<td><strong>Spec Operations</strong></td>
<td>extra argument (newer spec)</td>
<td>compliant</td>
<td>compliant</td>
</tr>
<tr>
<td><strong>Key Declaration</strong></td>
<td>/ @key</td>
<td>single</td>
<td>pair of</td>
</tr>
<tr>
<td><strong>Required App. IDs</strong></td>
<td>none</td>
<td>publisher</td>
<td>none</td>
</tr>
<tr>
<td><strong>Transport Config</strong></td>
<td>none</td>
<td>file-based or code-based</td>
<td>code-based</td>
</tr>
</tbody>
</table>
## Portability Challenges

<table>
<thead>
<tr>
<th></th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainParticipant</td>
<td>compliant</td>
<td>proprietary function</td>
<td></td>
</tr>
<tr>
<td>Factory compliant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Data Types</td>
<td>static method</td>
<td>member method</td>
<td>member method</td>
</tr>
<tr>
<td>Spec Operations</td>
<td>extra argument (newer spec)</td>
<td>compliant</td>
<td>compliant</td>
</tr>
<tr>
<td>Key Declaration</td>
<td>//@key uppercase</td>
<td>single #pragma</td>
<td>pair of #pragma</td>
</tr>
<tr>
<td></td>
<td>struct Info { long id; //@key</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>string msg; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>#pragma keylist list Info id</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>#pragma DCPS_DATA_TYPE &quot;Info&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>#pragma DCPS_DATA_KEY &quot;id&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>publisher &amp; subscriber</td>
<td>code-based</td>
<td>file-based or code-based</td>
</tr>
</tbody>
</table>
Lessons Learned - Pros

• DDS implementations are optimized for different use cases & design spaces
  – Low latency for collocated publishers and subscribers
Lessons Learned - Pros

• DDS implementations are optimized for different use cases & design spaces
  - Low latency for collocated publishers and subscribers
  - Low latency for remote publishers and subscribers
Lessons Learned - Pros

• DDS implementations are optimized for different use cases & design spaces
  – Low latency for collocated publishers and subscribers
  – Low latency for remote publishers and subscribers
  – Scalability of the number of subscribers
Lessons Learned - Cons

- Can’t yet make “apples-to-apples” DDS test parameters comparison for all impls
  - No common transport protocol
    - DDS1 uses RTPS on top of UDP (RTPS support planned this winter for DDS2)
    - DDS3 uses raw TCP or UDP
    - Centralized/Federated/Decentralized Architectures
  - Broadcast can be a two-edged sword (router overload!)
- DDS applications not yet portable “out-of-the-box”
  - New, rapidly evolving spec
  - Vendors use proprietary techniques to fill gaps, optimize
  - Clearly a need for portability wrapper facades, a la ACE or IONA’s POA utils
  - Lots of tuning & tweaking of policies & options are required to optimize performance
Lessons Learned - Cons

- Can’t yet make “apples-to-apples” DDS test parameters comparison for all impls
  - No common transport protocol
    - DDS1 uses RTPS on top of UDP (RTPS support planned this winter for DDS2)
    - DDS3 uses raw TCP or UDP
  - Centralized/Federated/Decentralized Architectures
- Broadcast can be a two-edged sword (router overload!)
- DDS applications not yet portable “out-of-the-box”
  - New, rapidly evolving spec
  - Vendors use proprietary techniques to fill gaps, optimize
  - Clearly a need for portability wrapper facades, a la ACE or IONA’s POA utils
- Lots of tuning & tweaking of policies & options are required to optimize performance
Future Work - Pub/Sub Metrics

• Tailor benchmarks to explore key classes of tactical applications
  • e.g., command & control, targeting, route planning
• Devise generators that can emulate various workloads & use cases
• Include wider range of QoS & configuration, e.g.:
  • Durability
  • Reliable vs best effort
  • Interaction of durability, reliability and history depth
  • Complementing of transport priority & latency budget (urgency)
• Measure migrating processing to source
• Measure discovery time for various entities
  • e.g., subscribers, publishers, & topics
• Find scenarios that distinguish performance of QoS policies & features, e.g.:
  • Listener vs waitset
  • Collocated applications
  • Very large # of subscribers & payload sizes
Future Work - Pub/Sub Metrics

- Tailor benchmarks to explore key classes of tactical applications
  - e.g., command & control, targeting, route planning
- Devise generators that can emulate various workloads & use cases
- Include wider range of QoS & configuration, e.g.:
  - Durability
  - Reliable vs best effort
  - Interaction of durability, reliability and history depth
  - Map to classes of tactical applications
- Measure migrating processing to source
- Measure discovery time for various entities
  - e.g., subscribers, publishers, & topics
- Find scenarios that distinguish performance of QoS policies & features, e.g.:
  - Listener vs waitset
  - Collocated applications
  - Very large # of subscribers & payload sizes
Future Work - Benchmarking Framework

• Larger, more complex automated tests
  • More nodes
  • More publishers, subscribers per test, per node
• Variety of data sizes, types
• Multiple topics per test
• Dynamic tests
  • Late-joining subscribers
  • Changing QoS values
• Alternate throughput measurement strategies
  • Fixed # of samples – measure elapsed time
  • Fixed time window – measure # of samples
  • Controlled publish rate
• Generic testing framework
  • Common test code
  • Wrapper facades to factor out portability issues
• Include other pub/sub platforms
  • WS Notification
  • ICE pub/sub
  • Java impls of DDS

DDS benchmarking framework is open-source & available on request
Future Work - Benchmarking Framework

- Larger, more complex automated tests
  - More nodes
  - More publishers, subscribers per test, per node
- Variety of data sizes, types
- Multiple topics per test
- Dynamic tests
  - Late-joining subscribers
  - Changing QoS values
- Alternate throughput measurement strategies
  - Fixed # of samples – measure elapsed time
  - Fixed time window – measure # of samples
  - Controlled publish rate
- Generic testing framework
  - Common test code
  - Wrapper facades to factor out portability issues
- Include other pub/sub platforms
  - WS Notification
  - ICE pub/sub
  - Java impls of DDS

DDS benchmarking framework is open-source & available on request
Concluding Remarks

• Next-generation QoS-enabled information management for tactical applications requires innovations & advances in tools & platforms

• Emerging COTS standards address some, but not all, hard issues!

• These benchmarks are a snapshot of an ongoing process

• Keep track of our benchmarking work at www.dre.vanderbilt.edu/DDS

• Latest version of these slides at DDS_RTWS06.pdf in the above directory

Thanks to OCI, PrismTech, & RTI for providing their DDS implementations & for helping with the benchmark process