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

Jeff Parsons
j.parsons@vanderbilt.edu

Ming Xiong
xiongm@isis.vanderbilt.edu

Dr. Douglas C. Schmidt
d.schmidt@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
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)
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><strong>Notification Mechanism</strong></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><strong>Transport</strong></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><strong>Higher-level DDS Protocol</strong></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><strong>Lower-level Transport</strong></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 LOCAL TRANSIENT 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
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
// Complex Sequence Type
struct Inner {
    string info;
    long index;
};

typedef sequence<Inner> InnerSeq;
struct Outer {
    long length;
    InnerSeq nested_member;
};

typedef sequence<Outer> ComplexSeq;

DDS vs Other Pub/Sub Architectures

Measured avg. round-trip latency & jitter
100 primer samples
10,000 stats samples

Process 1
Blade 0

Tested seq. of byte & seq. of complex type

Process 2
Blade 0

Ack message of 4 bytes

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

X & Y axes of all graphs in presentation use log scale for readability
Latency – Simple Data Type

DDS/GSOAP/JMS/Notification Service Comparison - Latency

- DDS1
- DDS2
- DDS3
- GSOAP
- JMS
- Notification Service

Message Length (samples)
Latency – Simple Data Type

With conventional pub/sub mechanisms the delay before the application learns critical information is very high!

In contrast, DDS latency is low across the board.
Jitter – Simple Data Type

DDS/GSOAP/JMS/Notification Service Comparison - Jitter

Message Length (samples)

Standard Deviation (usecs)
Conventional pub/sub mechanisms exhibit extremely high jitter, which makes them unsuitable for tactical systems.

In contrast, DDS jitter is low across the board.
Latency – Complex Data Type

DDS/GSOAP Comparison - Complex Type Latency

- DDS1
- DDS2
- DDS3
- GSOAP

Avg. Latency (usecs)

Message Length (samples)
While latency with complex types is less flat for all, DDS still scales better than Web Services by a factor of 2 or more.

Some DDS implementations optimized for smaller data sizes.
Jitter – Complex Data Type

DDS/GSOAP Comparison - Complex Type Jitter

- DDS1
- DDS2
- DDS3
- GSOAP

Standard Deviation (usecs)

Message Length (samples)
Measuring jitter with complex data types brings out even more clearly the difference between DDS & Web Services. Better performance can be achieved by optimizing for certain data sizes.
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

Blade 0

Blade N

Blade ...

Blade 2

Blade 1

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

Performance levels off less than for unicast

- subscriber uses listener
- no daemon (library per node)
- KEEP_LAST (depth = 1)
Scaling Up Subscribers – DDS1 1 to 4

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)
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)
Impl Comparison: 12 Subscribers Multicast

- DDS1
- DDS2

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

 DDS1  DDS3

49
Impl Comparison: 12 Subscribers Unicast

Performance differences slightly less than with 8 subscribers

Unicast not supported by DDS2

- subscriber uses listener
- KEEP_ALL

Bytes

DDS1

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

Waitset

Condition
Condition
Condition

Data Reader

DDS

take_w_condition()

wait()

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 -> Blade 4
Blade 0 -> Blade 3
Blade 0 -> Blade 2
Blade 0 -> Blade 1

Byte sequences

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

100 primer samples
10,000 stats samples

Blade 0
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

```
Publisher

Subscriber

KEEP_ALL
KEEP_LAST = 10
MAX_SAMPLES = 5
MAX_SAMPLES = 5

Subscriber

DDS

Transport

DDS
```

```
## 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>compliant</td>
<td>proprietary function</td>
</tr>
<tr>
<td>Factory</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</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</td>
<td>compliant</td>
<td>compliant</td>
<td>proprietary function</td>
</tr>
<tr>
<td>Factory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Data Types</td>
<td>member function</td>
<td>member function</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></td>
<td>single pair of</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>#pragma</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pair of #pragma</td>
<td></td>
</tr>
<tr>
<td>Required App. IDs</td>
<td>publisher &amp; subscriber</td>
<td>none</td>
<td>publisher</td>
</tr>
<tr>
<td>Required App.</td>
<td>code-based</td>
<td>none</td>
<td>file-based or code-based</td>
</tr>
<tr>
<td>Transport Config</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Portability Challenges

<table>
<thead>
<tr>
<th>DomainParticipant Factory</th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
</tr>
</thead>
<tbody>
<tr>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>extra argument</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Declaration</td>
<td>//@key</td>
<td>single</td>
<td>pair of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pragma</td>
</tr>
<tr>
<td>Required App. IDs</td>
<td>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>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>create_publisher(##)</td>
<td>single</td>
<td>pair of</td>
</tr>
<tr>
<td>Required App. IDs</td>
<td>none</td>
<td>publisher</td>
<td>file-based or code-based</td>
</tr>
<tr>
<td>Transport Config</td>
<td>none</td>
<td>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 Factory compliant</td>
<td>compliant</td>
<td>proprietary function</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</td>
<td>single #pragma</td>
<td>pair of #pragma</td>
</tr>
<tr>
<td>struct Info {</td>
<td>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></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>
</tbody>
</table>

```c
struct Info {
    long id; //@key
    string msg;
};

//@key
```
Lessons Learned - Pros

- Performance of DDS is significantly faster than other pub/sub architectures
  - Even the slowest was 2x faster than other pub/sub services
- DDS scales better to larger payloads, especially for simple data types

<table>
<thead>
<tr>
<th>Message Size (bytes)</th>
<th>DDS1</th>
<th>DDS2</th>
<th>DDS3</th>
<th>GSOAP</th>
<th>JMS</th>
<th>Notification Service</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg. Latency (usecs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>1024</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>2048</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>4096</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>8192</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>16384</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
</tbody>
</table>
Lessons Learned - Pros

- Performance of DDS is significantly faster than other pub/sub architectures
  - Even the slowest was 2x faster than other pub/sub services
- DDS scales better to larger payloads, especially for simple data types
- DDS implementations are optimized for different use cases & design spaces
  - e.g., smaller/larger payloads & smaller/larger # 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
  • Unicast/Broadcast/Multicast

<table>
<thead>
<tr>
<th>Impl</th>
<th>unicast</th>
<th>multicast</th>
<th>broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS1</td>
<td>Yes (default)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DDS2</td>
<td>No</td>
<td>Yes</td>
<td>Yes (default)</td>
</tr>
<tr>
<td>DDS3</td>
<td>Yes (default)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

• Centralized/Federated/Decentralized Architectures

• 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
• Broadcast can be a two-edged sword (router overload!)
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
  - Unicast/Broadcast/Multicast
    - DDS1: Yes (default)
    - DDS2: No
    - DDS3: Yes (default)

<table>
<thead>
<tr>
<th>Impl</th>
<th>unicast</th>
<th>multicast</th>
<th>broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS1</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DDS2</td>
<td>No</td>
<td>Yes</td>
<td>Yes (default)</td>
</tr>
<tr>
<td>DDS3</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- Centralized/Federated/Decentralized Architectures

- 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
  - Broadcast can be a two-edged sword (router overload!)
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 - 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