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

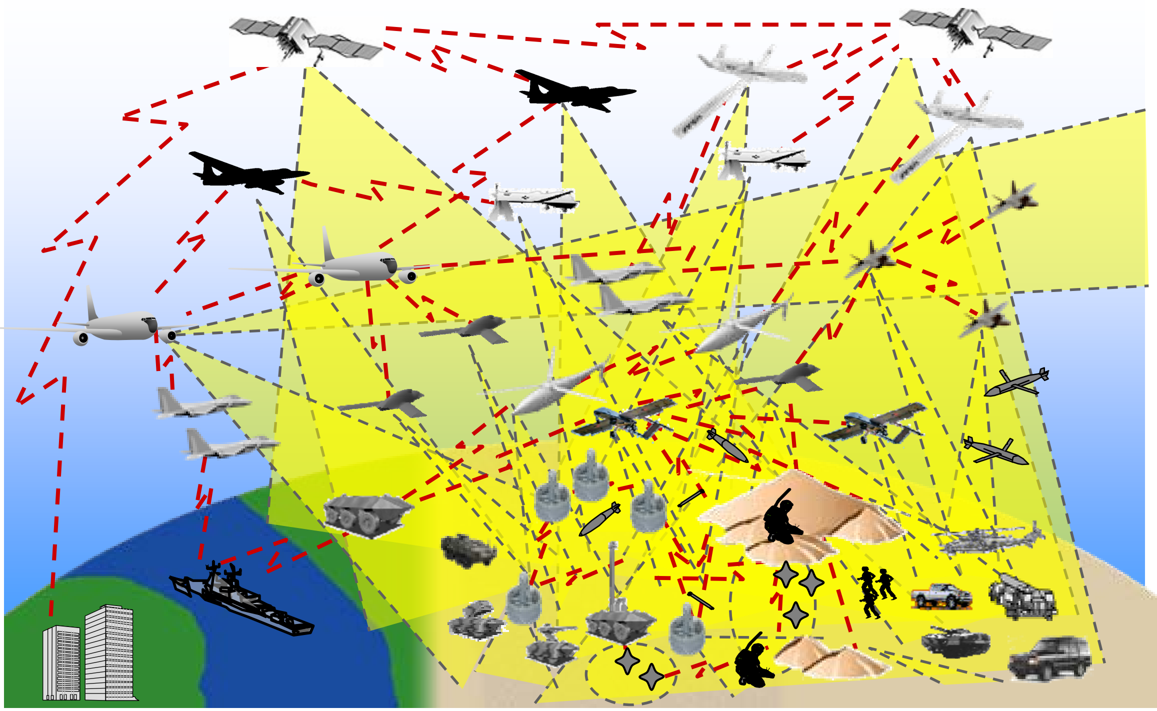
Demands on Tactical Information Svstems

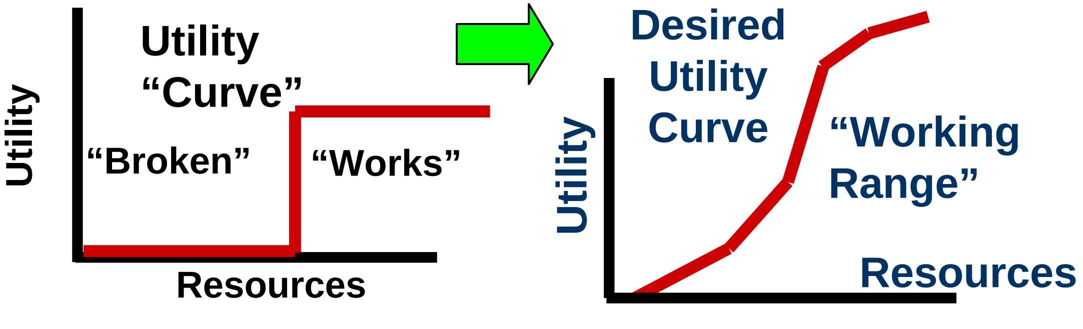
Key ***problem space*** challenges

- 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

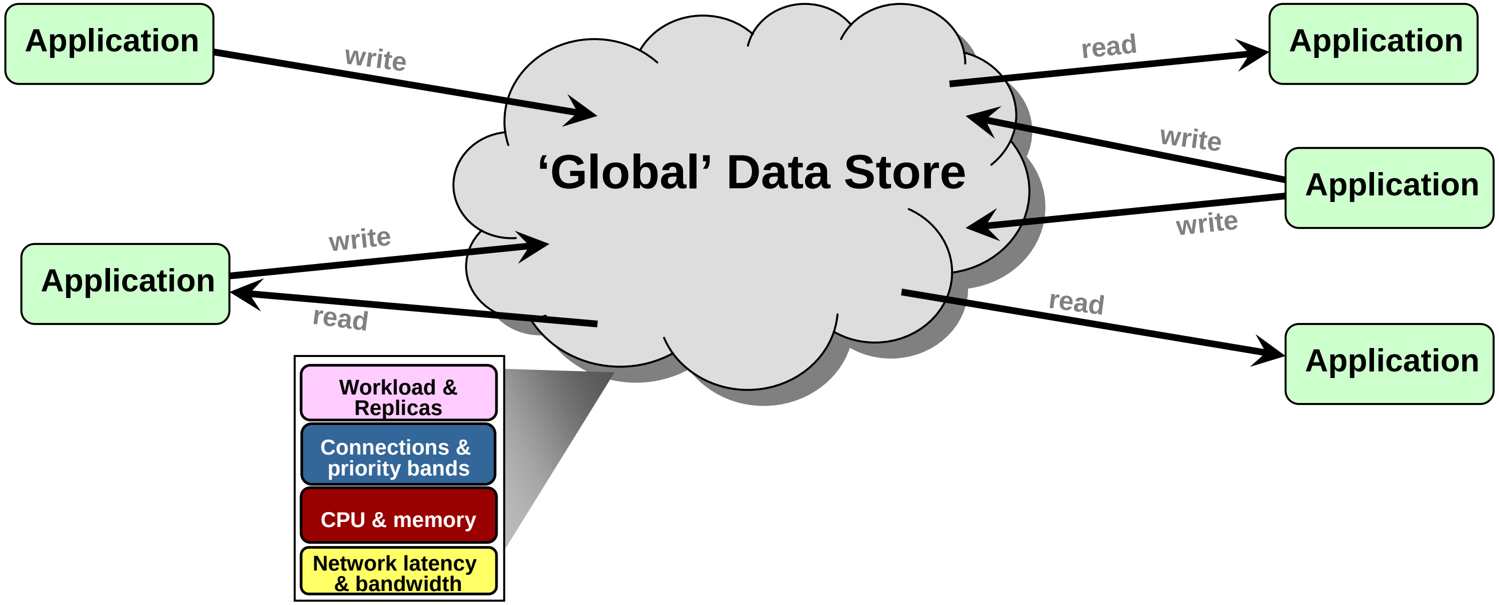




"Harder" Requirements

"Softer" Requirements

Promising Approach: The OMG Data Distribution Service (DDS)

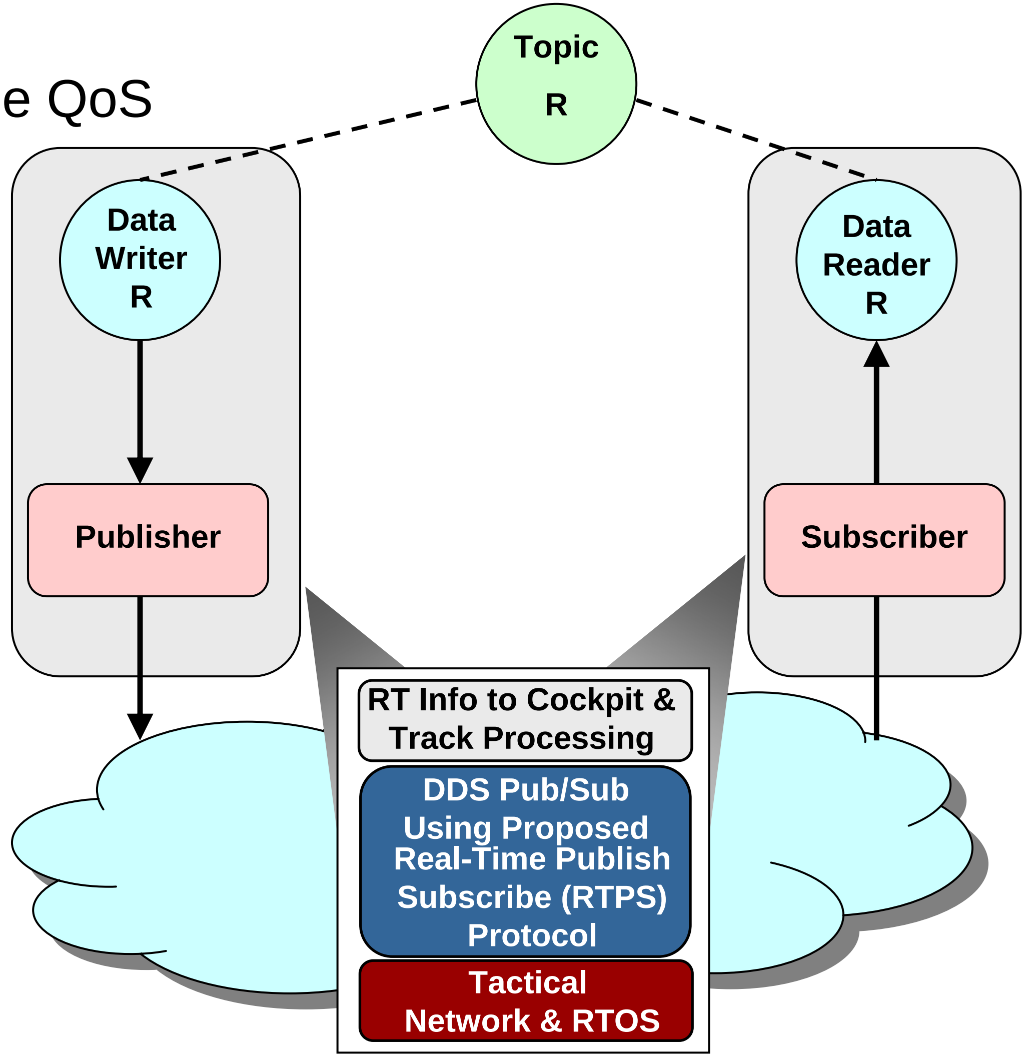


Provides flexibility, power & modular structure by decoupling:

- **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**

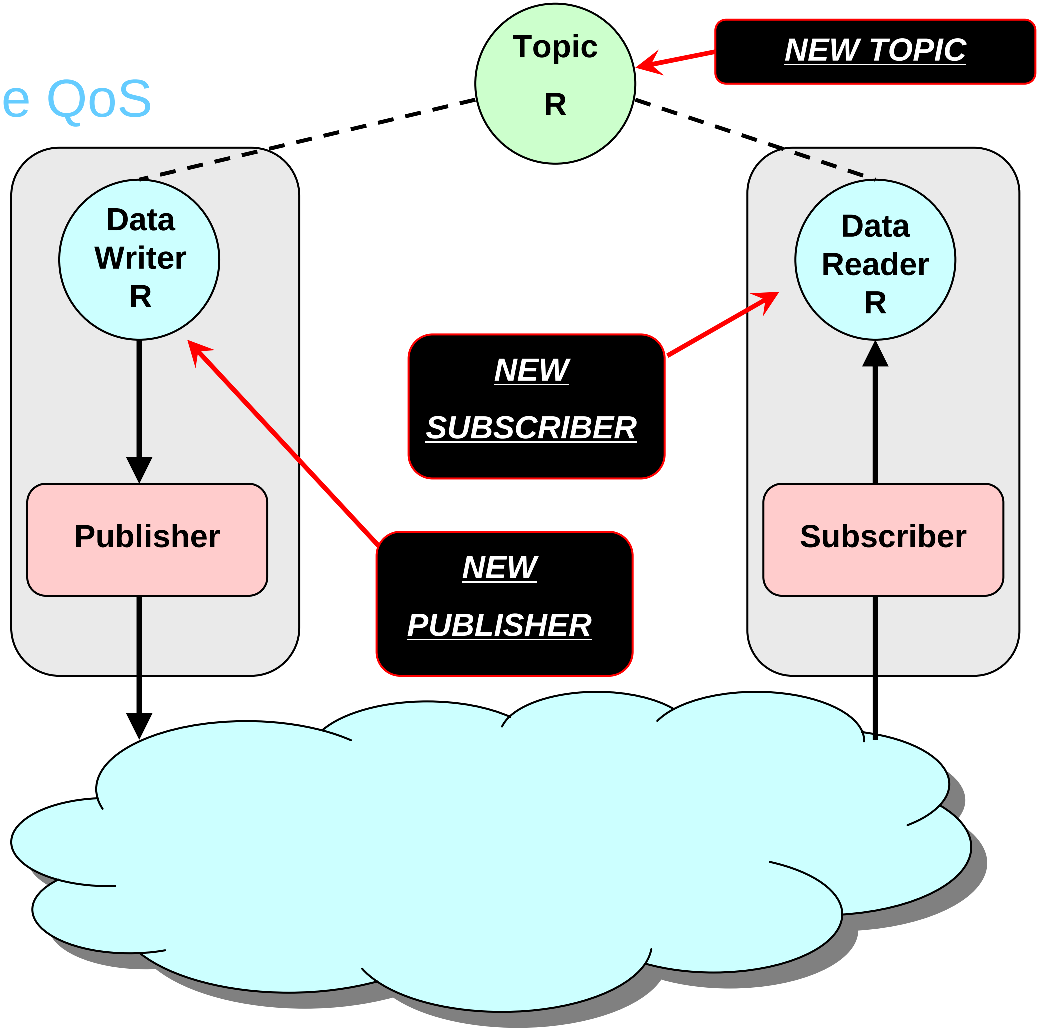
Overview of the Data Distribution Service (DDS)

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


Topic
R
Data
Writer
R
Publisher
Data
Reader
R
Subscriber
RT Info to Cockpit & Track Processing
DDS Pub/Sub Using Proposed Real-Time Publish Subscribe (RTPS) Protocol
Tactical Network & RTOS

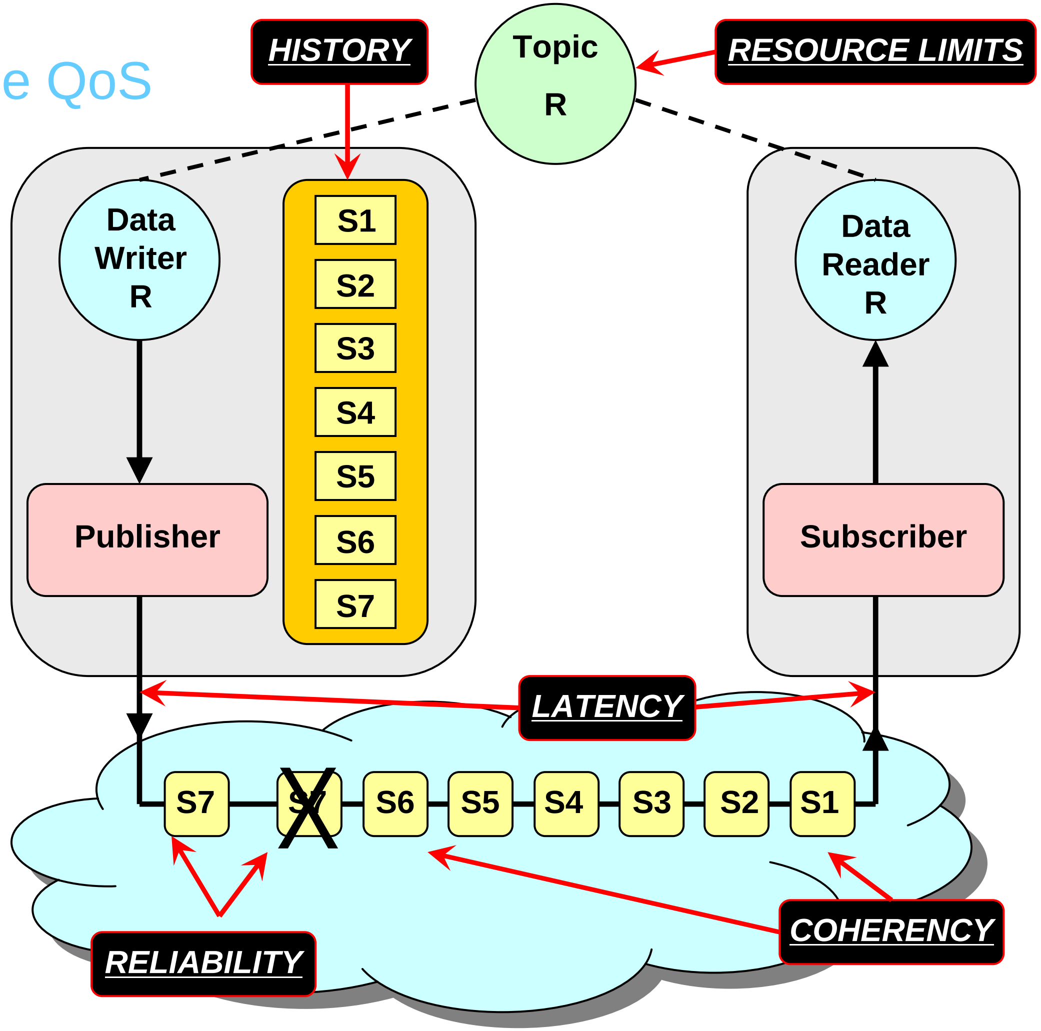
Overview of the Data Distribution Service (DDS)

- 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



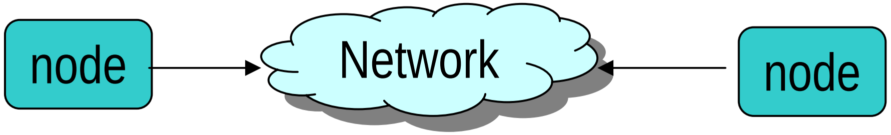
Overview of the Data Distribution Service (DDS)

- 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



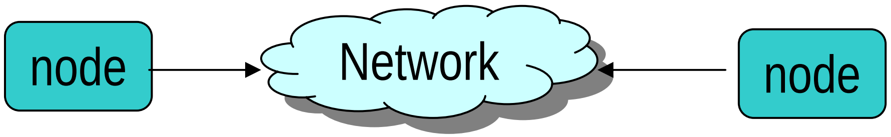
Overview of DDS Implementation Architectures

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

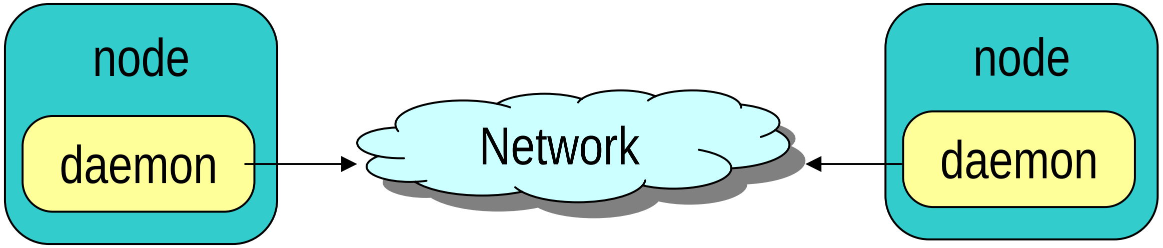


Overview of DDS Implementation Architectures

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

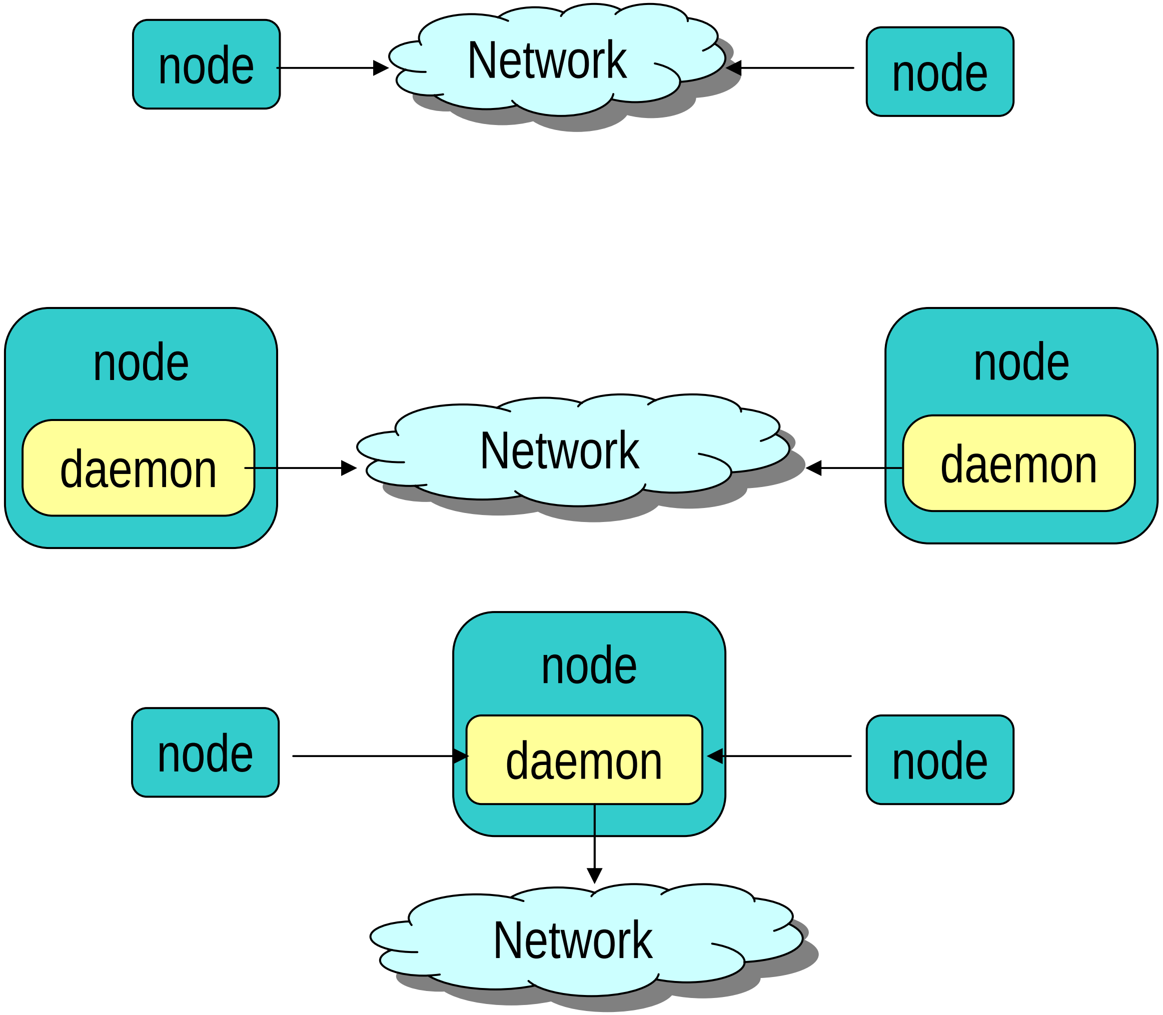


- **Federated Architecture**
  - a separate daemon process to handle communication, reliability, QoS, etc.

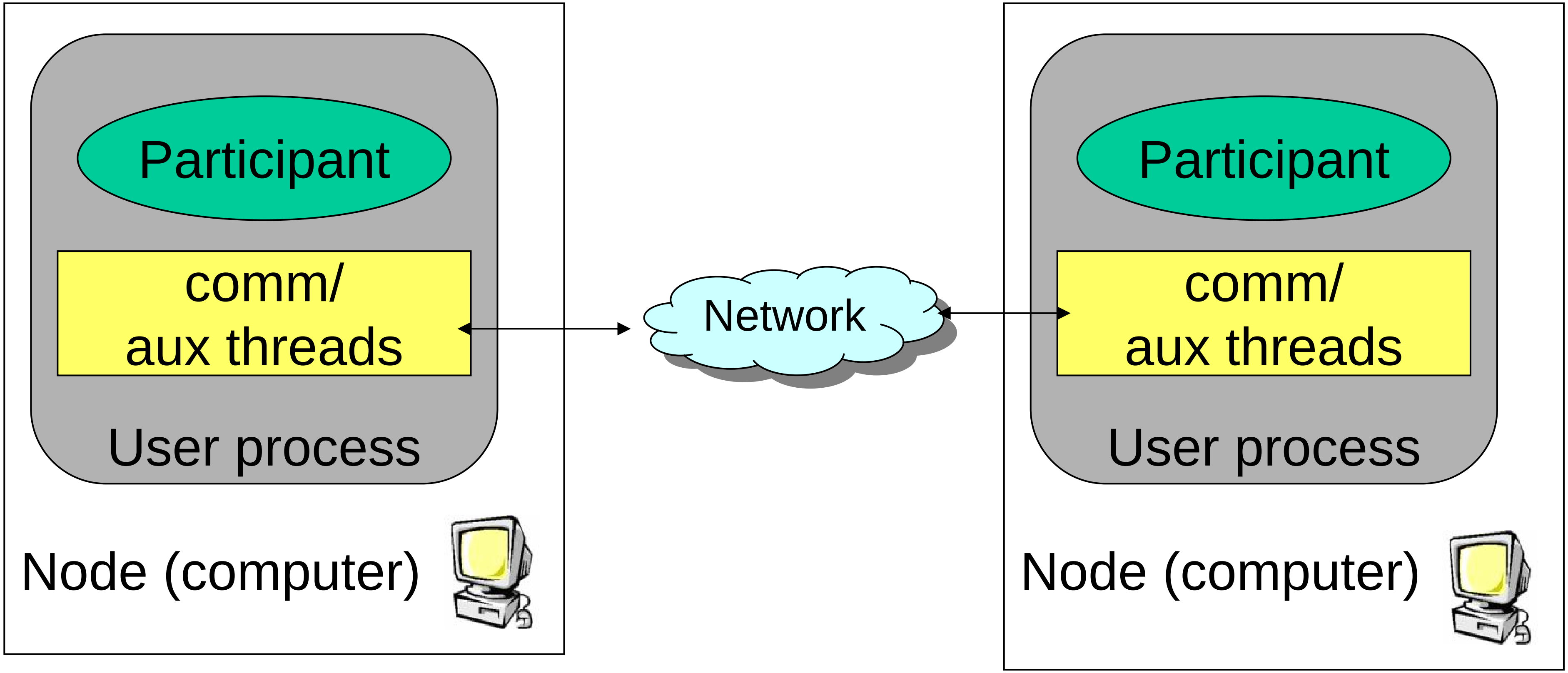


Overview of DDS Implementation Architectures

- **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

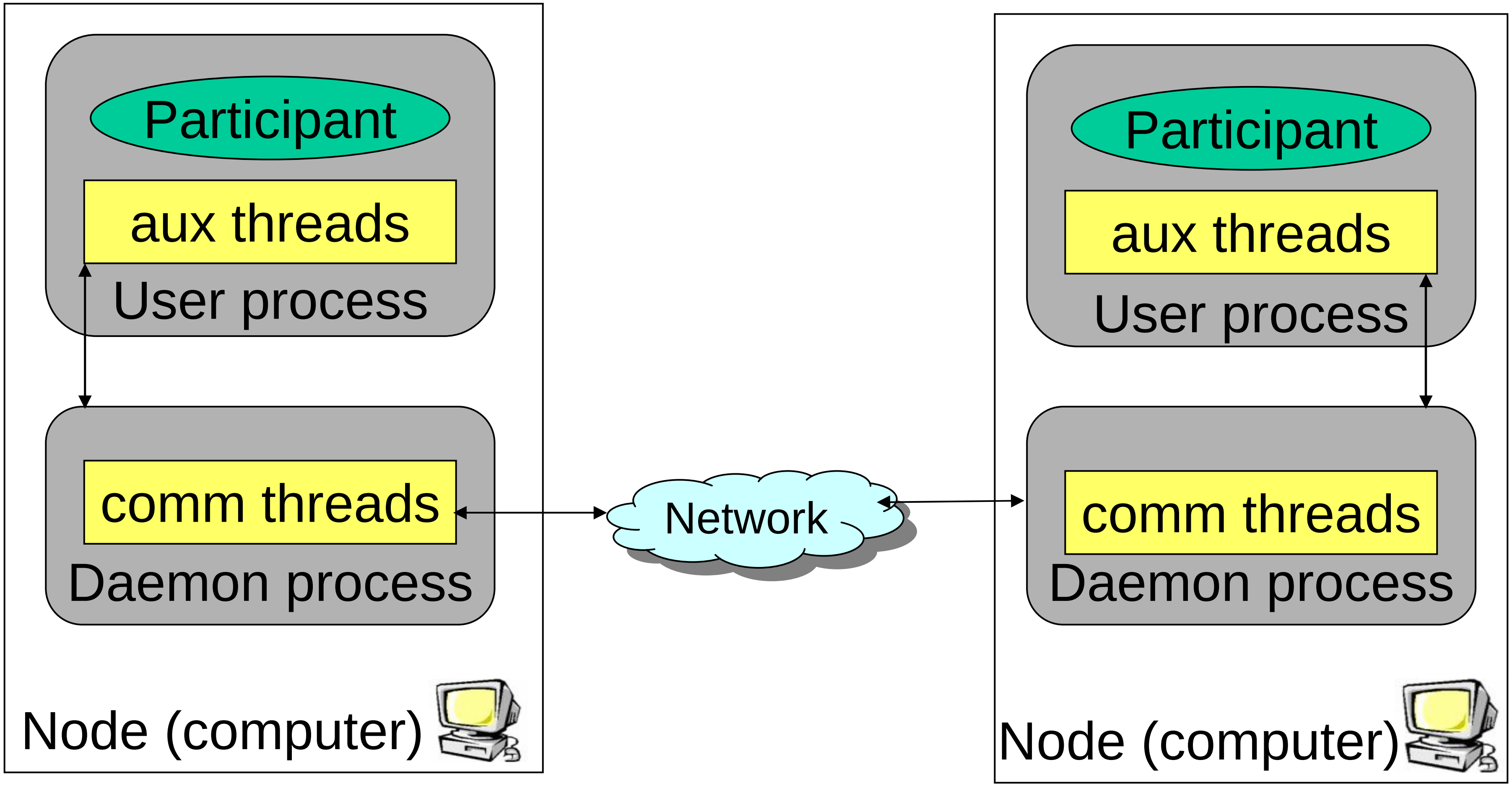


DDS1 (Decentralized Architecture)



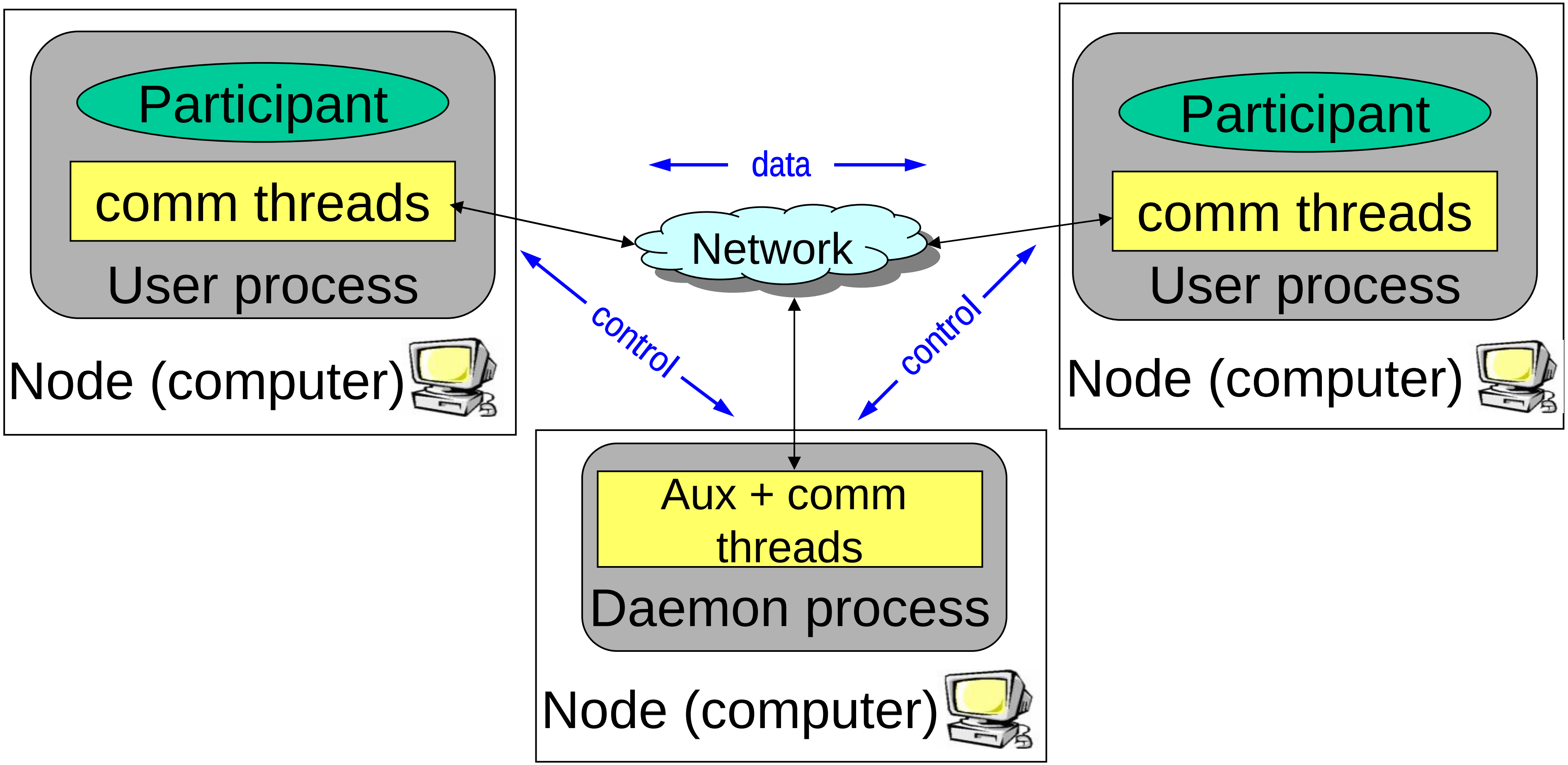
Pros: Self-contained communication end-points, needs no extra daemons
Cons: User process more complex, e.g., must handle config details (efficient discovery, multicast)

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

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

QoS Policies Comparison Table (partial)

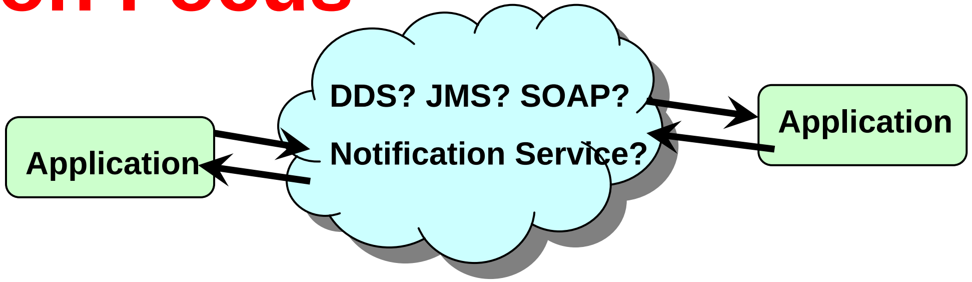
QoS	Description	DDS1	DDS2	DDS3
DURABILITY	Controls how long published samples are stored by the middleware for late-joining data readers	VOLATILE TRANSIENT-LOCAL	VOLATILE TRANSIENT-LOCAL TRANSIENT PERSISTENT	VOLATILE TRANSIENT_LOCAL
HISTORY	Sets number of samples that DDS will store locally for data writers & data readers	KEEP_LAST KEEP_ALL	KEEP_LAST KEEP_ALL	KEEP_LAST KEEP_ALL

RELIABILITY	Whether data published by a data writer will be reliably delivered by DDS to matching data readers	BEST_EFFORT
		RELIABLE
			BEST_EFFORT
			RELIABLE
				BEST_EFFORT(UDP)
				RELIABLE(TCP)

RESOURCE_LIMITS	Controls memory resources that DDS allocates & uses for data writer or data reader	initial_instance(extension) initial_samples(extension) max_instances max_samples max_samples_per_instance	max_instances max_samples max_samples_per_instance	max_instances max_samples max_samples_per_instance

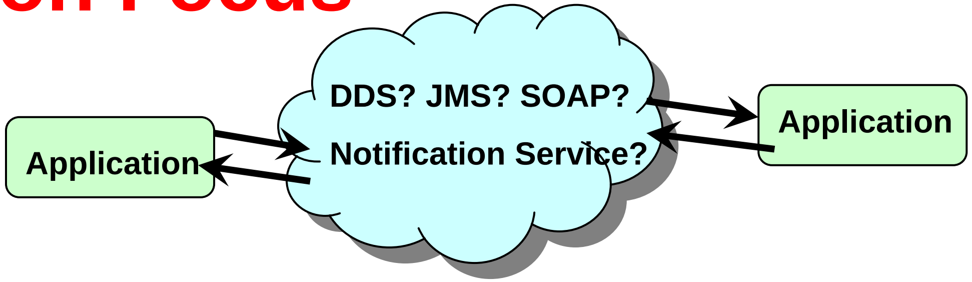
Evaluation Focus

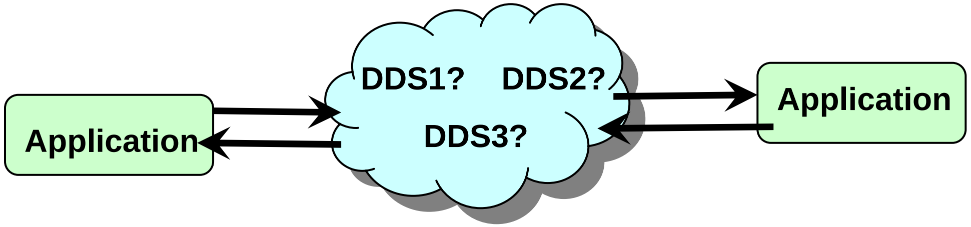
- Compare performance of C++ implementations of DDS to:
  - Other pub/sub middleware
    - CORBA Notification Service
    - SOAP
    - Java Messaging Service



Evaluation Focus

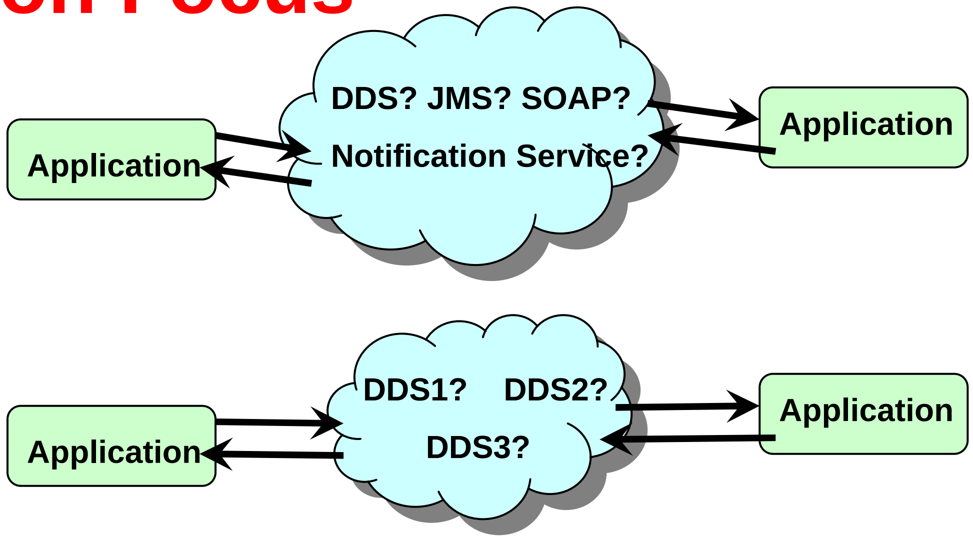
- Compare performance of C++ implementations of DDS to:
  - Other pub/sub middleware
    - CORBA Notification Service
    - SOAP
    - Java Messaging Service
  - Each other



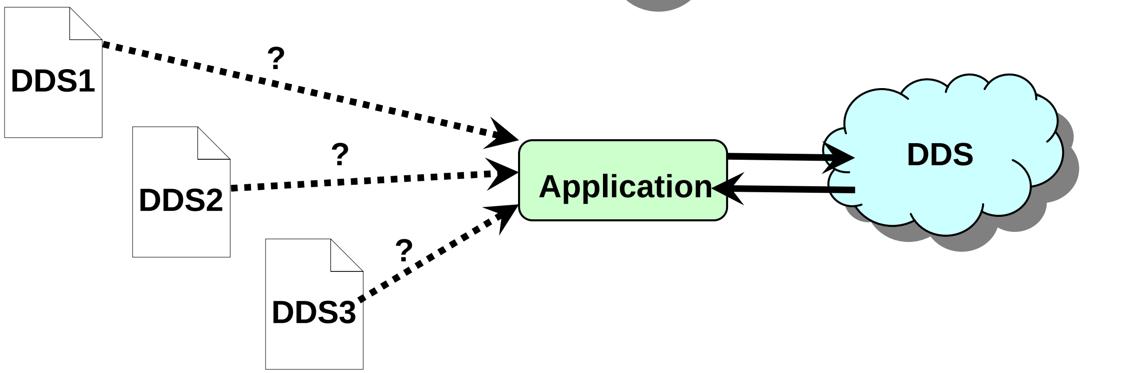


Evaluation Focus

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

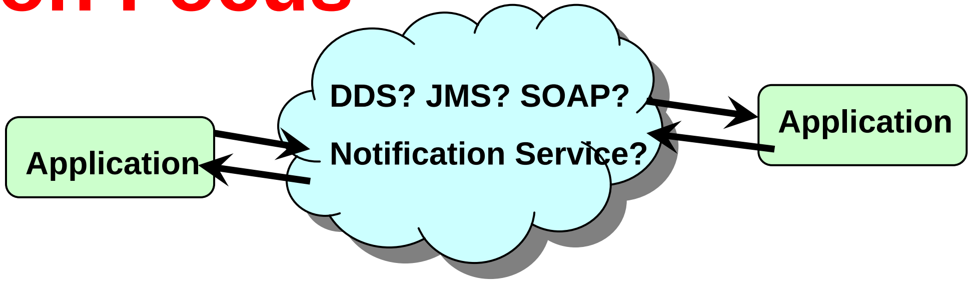


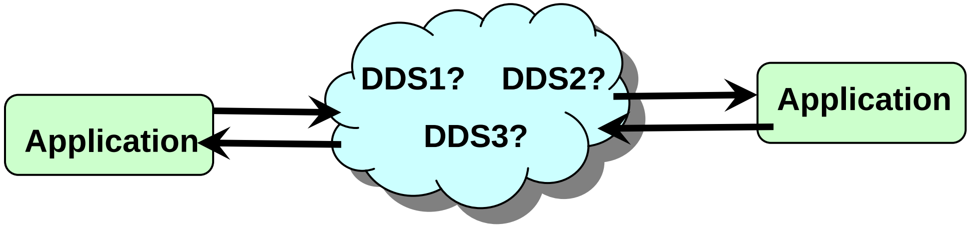
- Compare DDS portability & configuration details



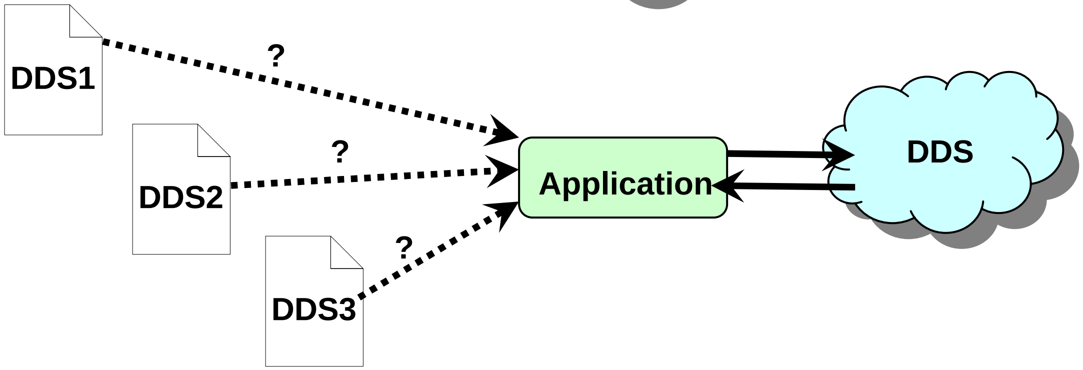
Evaluation Focus

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

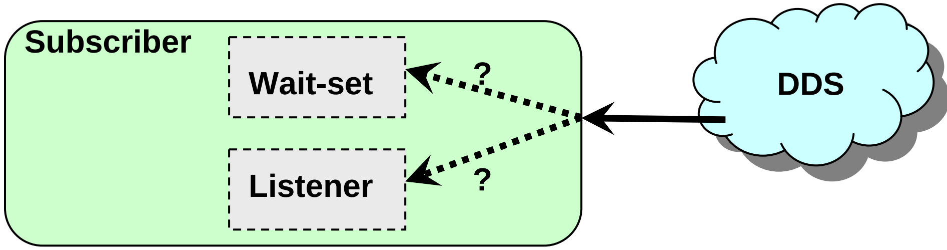




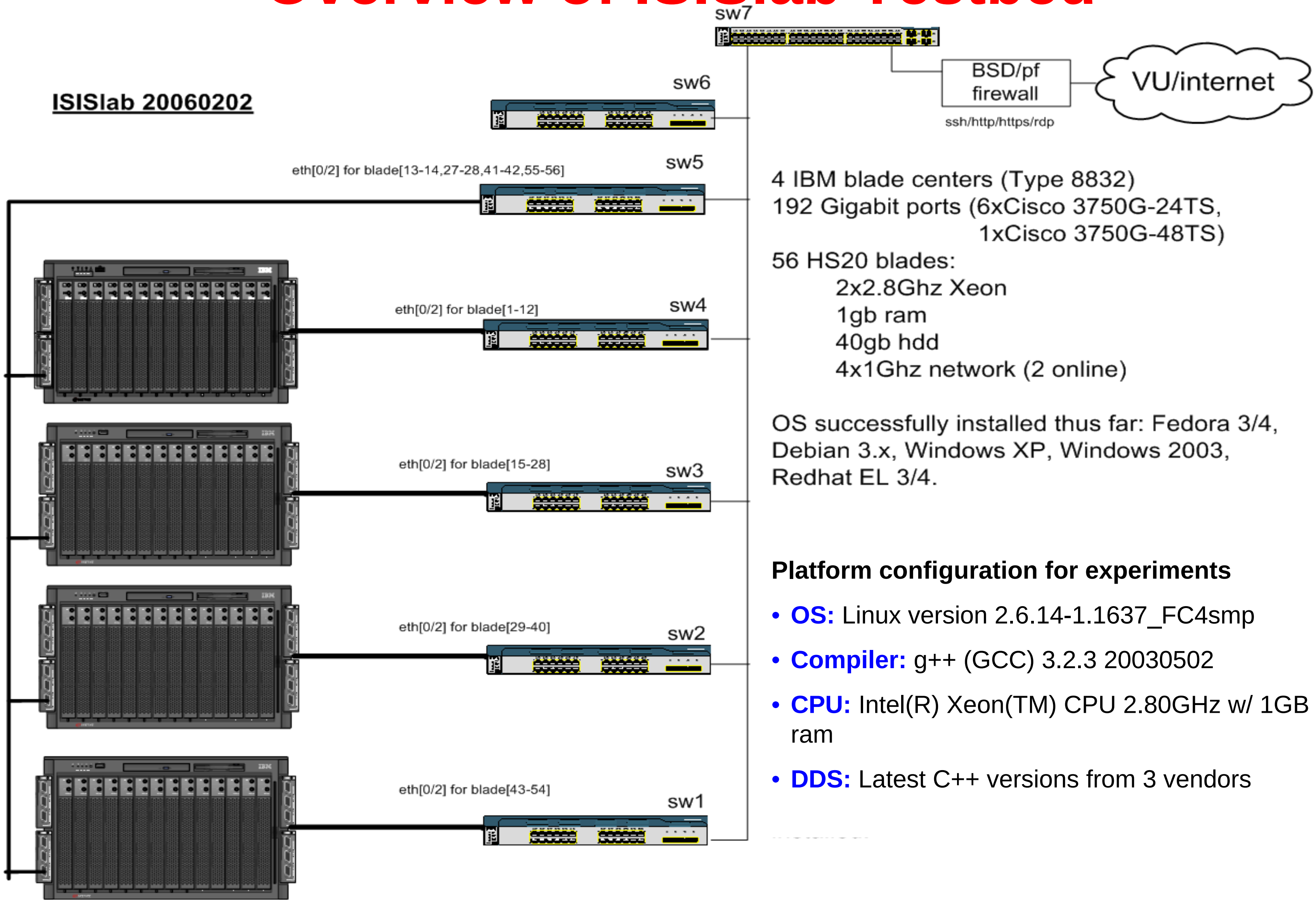
- Compare DDS portability & configuration details



- Compare performance of subscriber notification mechanisms
  - Listener vs. wait-set



Overview of ISISlab Testbed


sw7
BSD/pf firewall
VU/internet
ssh/http/https/rdp
ISISlab 20060202
sw6
sw5
eth[0/2] for blade[13-14,27-28,41-42,55-56]
sw4
eth[0/2] for blade[1-12]
sw3
eth[0/2] for blade[15-28]
sw2
eth[0/2] for blade[29-40]
sw1
eth[0/2] for blade[43-54]

4 IBM blade centers (Type 8832)
192 Gigabit ports (6xCisco 3750G-24TS,
1xCisco 3750G-48TS)
56 HS20 blades:
2x2.8Ghz Xeon
1gb ram
40gb hdd
4x1Ghz network (2 online)

OS successfully installed thus far: Fedora 3/4, Debian 3.x, Windows XP, Windows 2003, Redhat EL 3/4.

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

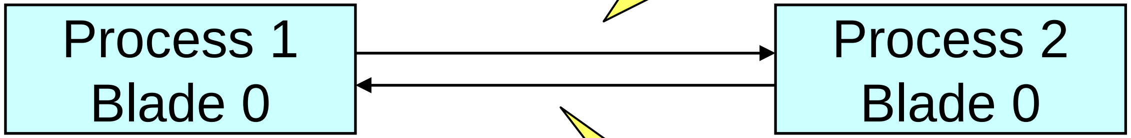
- Challenge – Measuring latency & throughput accurately without depending on synchronized clocks
- Solution
  - Latency – Add ack message, use publisher clock to time round trip
  - Throughput – Remove sample when read, use subscriber clock only
- Challenge – Managing many tests, payload sizes, nodes,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

DDS vs Other Pub/Sub Architectures

```
// Complex Sequence Type
struct Inner {
  string info;
  long index;
};
typedef sequence<Inner> InnerSeq;
struct Outer {
  long length;
  InnerSeq nested_member;
};
typedef sequence<Outer>
ComplexSeq;
```

Measured avg. round-trip latency & jitter

100 primer samples
10,000 stats samples



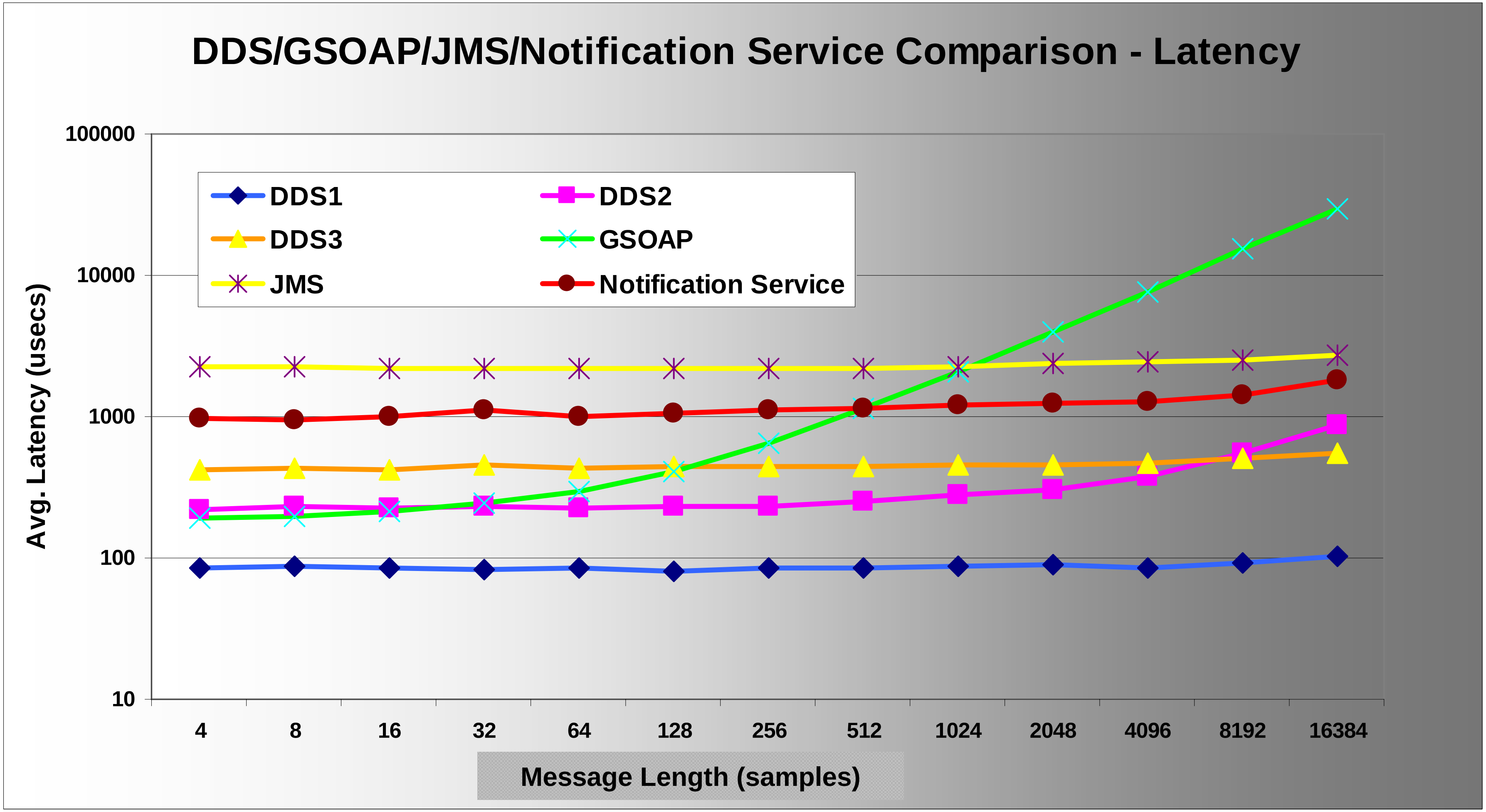
Ack message of 4 bytes

Tested seq. of byte & seq. of complex type

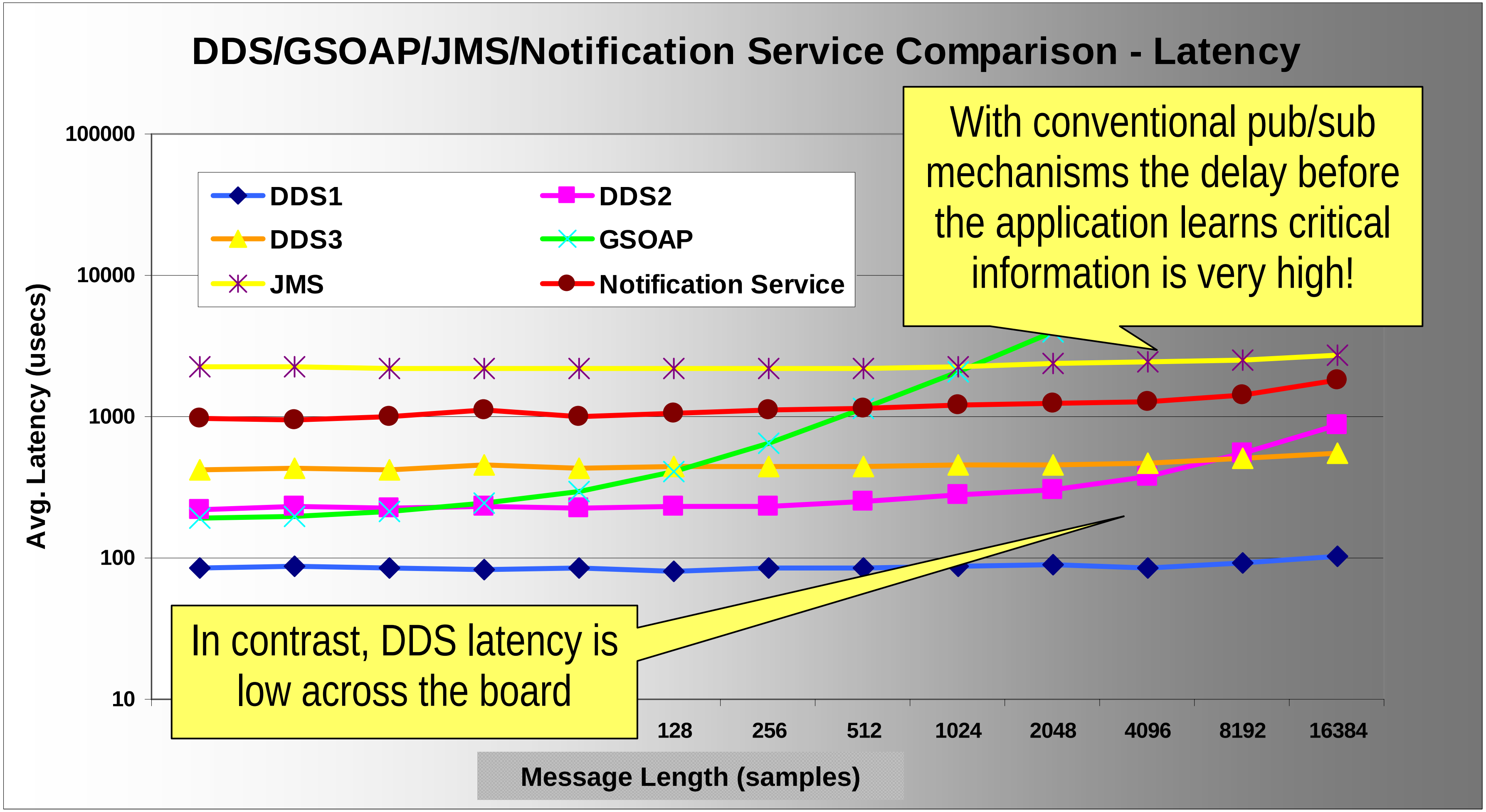
Seq. lengths in powers of 2 (4 – 16384)

X & Y axes of all graphs in presentation use log scale for readability

Latency – Simple Data Type



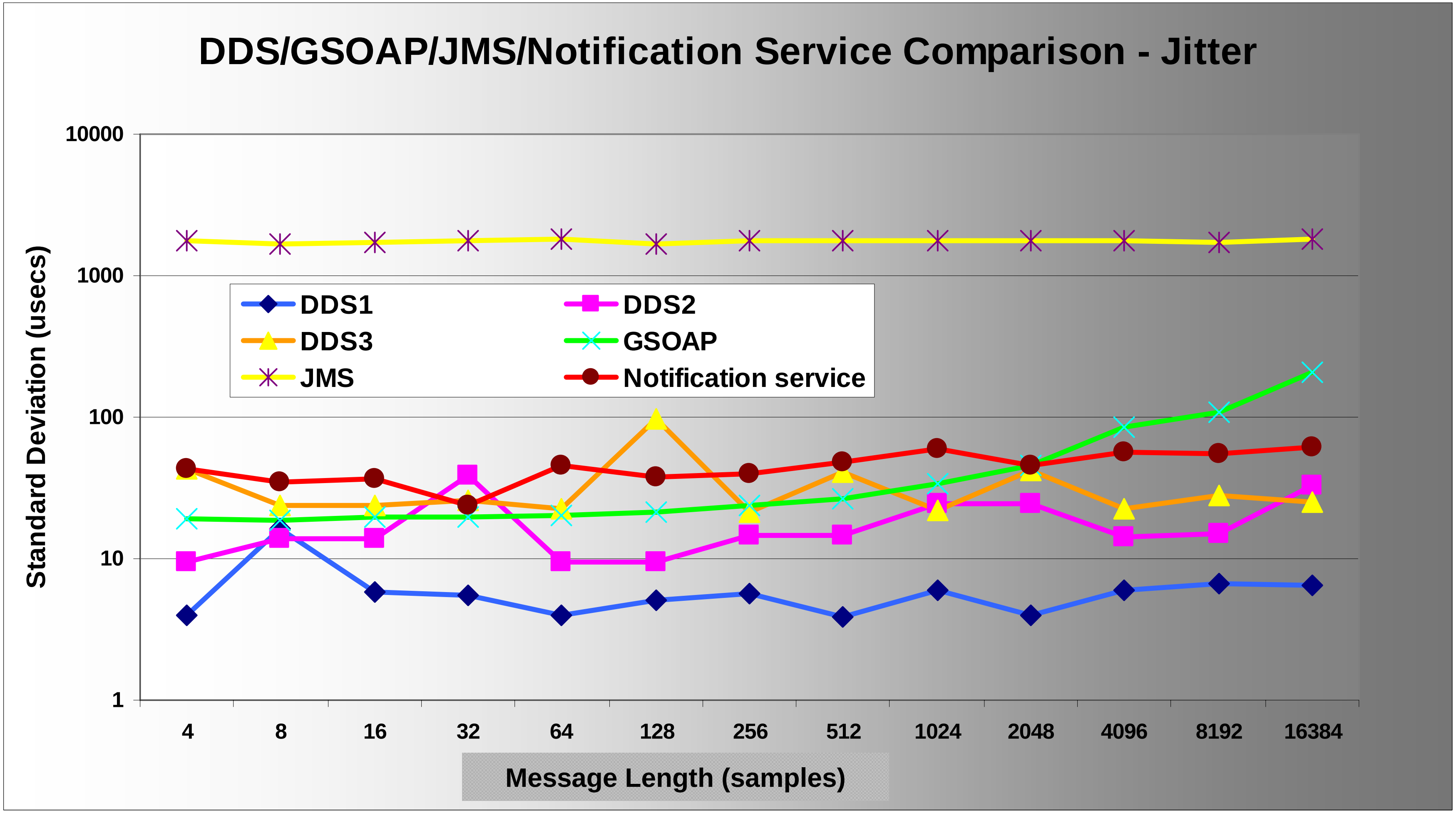
Latency – Simple Data Type



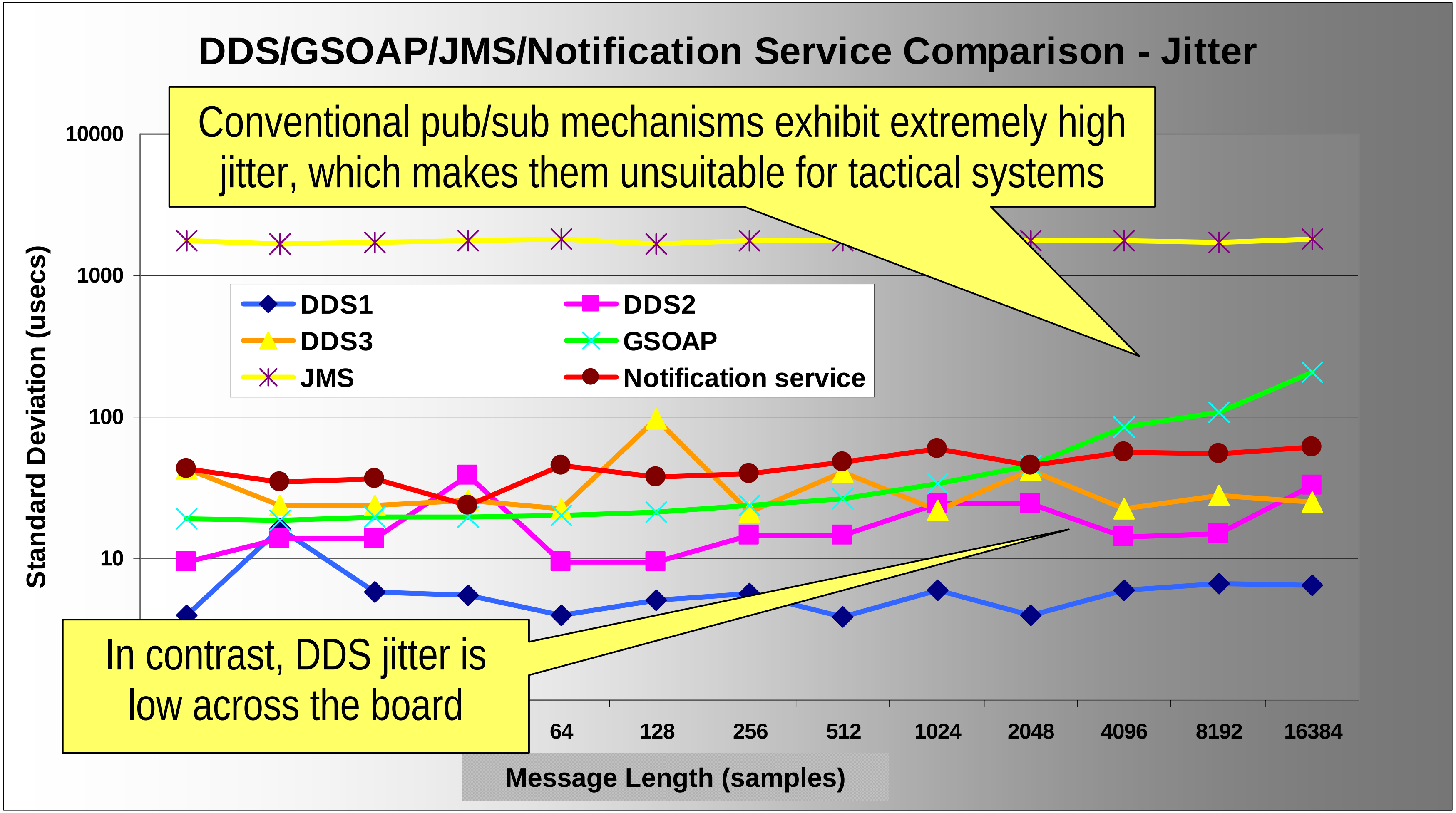
With conventional pub/sub mechanisms the delay before the application learns critical information is very high!

In contrast, DDS latency is low across the board

Jitter – Simple Data Type



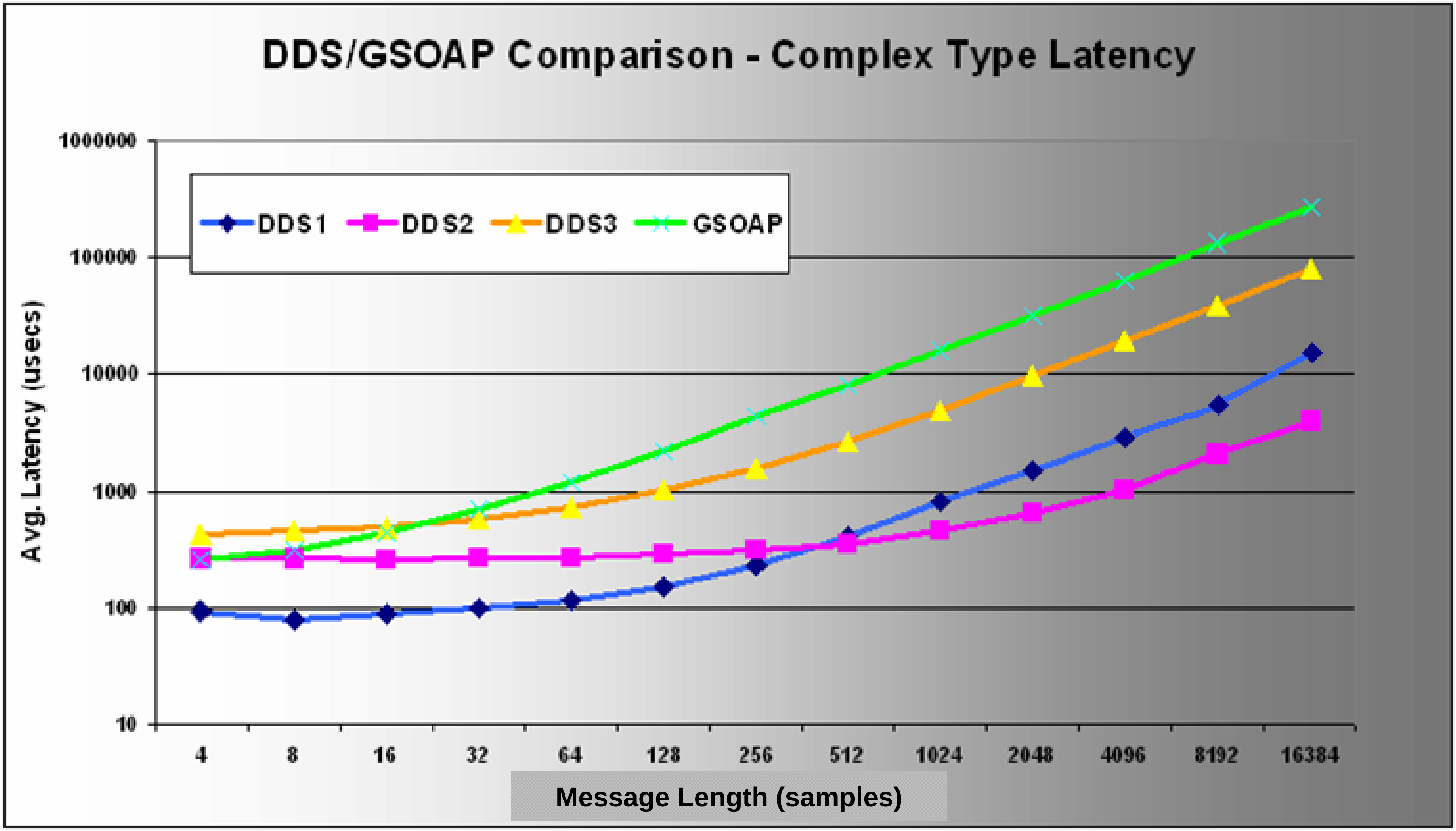
Jitter – Simple Data Type



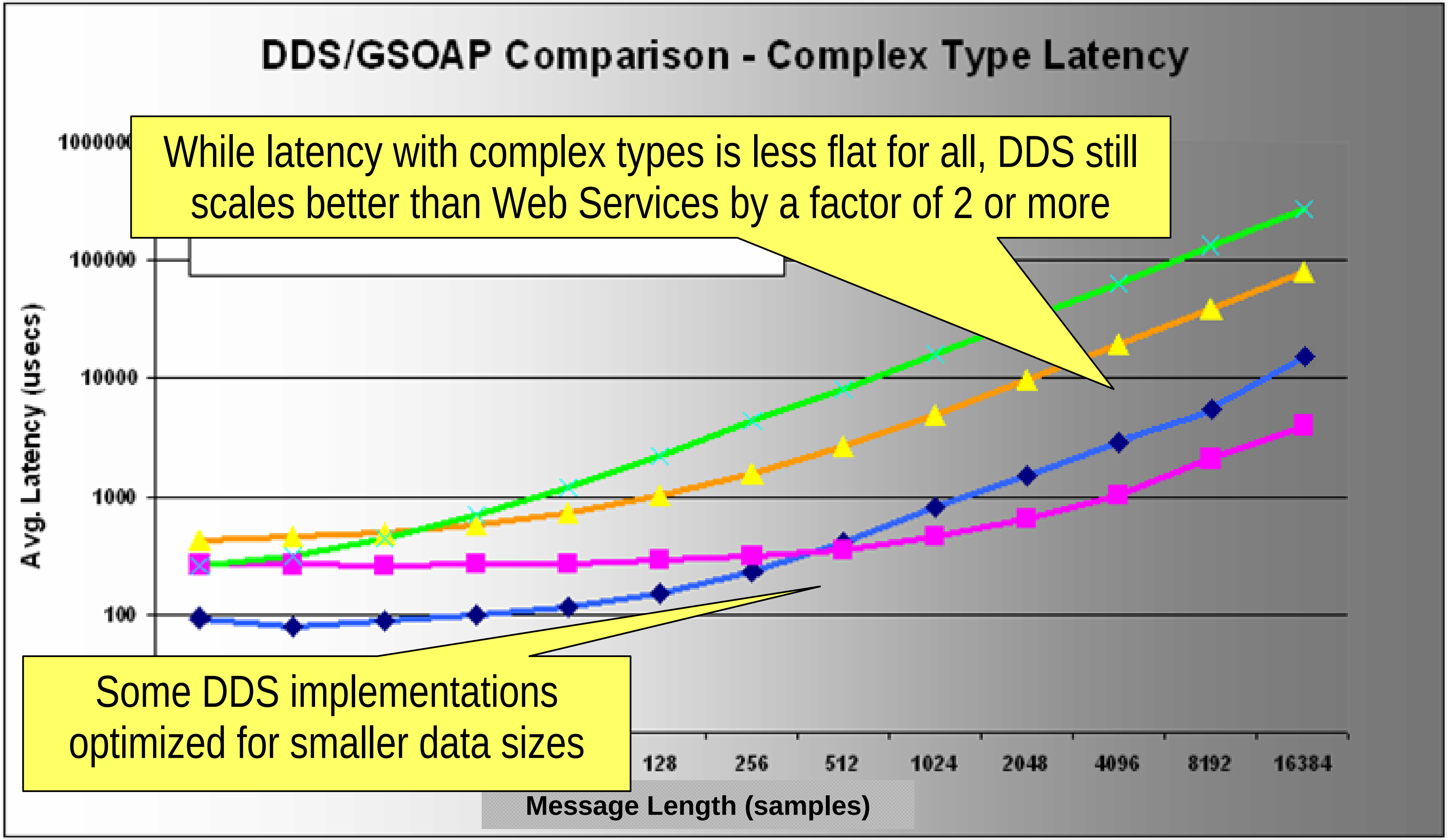
Conventional pub/sub mechanisms exhibit extremely high jitter, which makes them unsuitable for tactical systems

In contrast, DDS jitter is low across the board

Latency – Complex Data Type



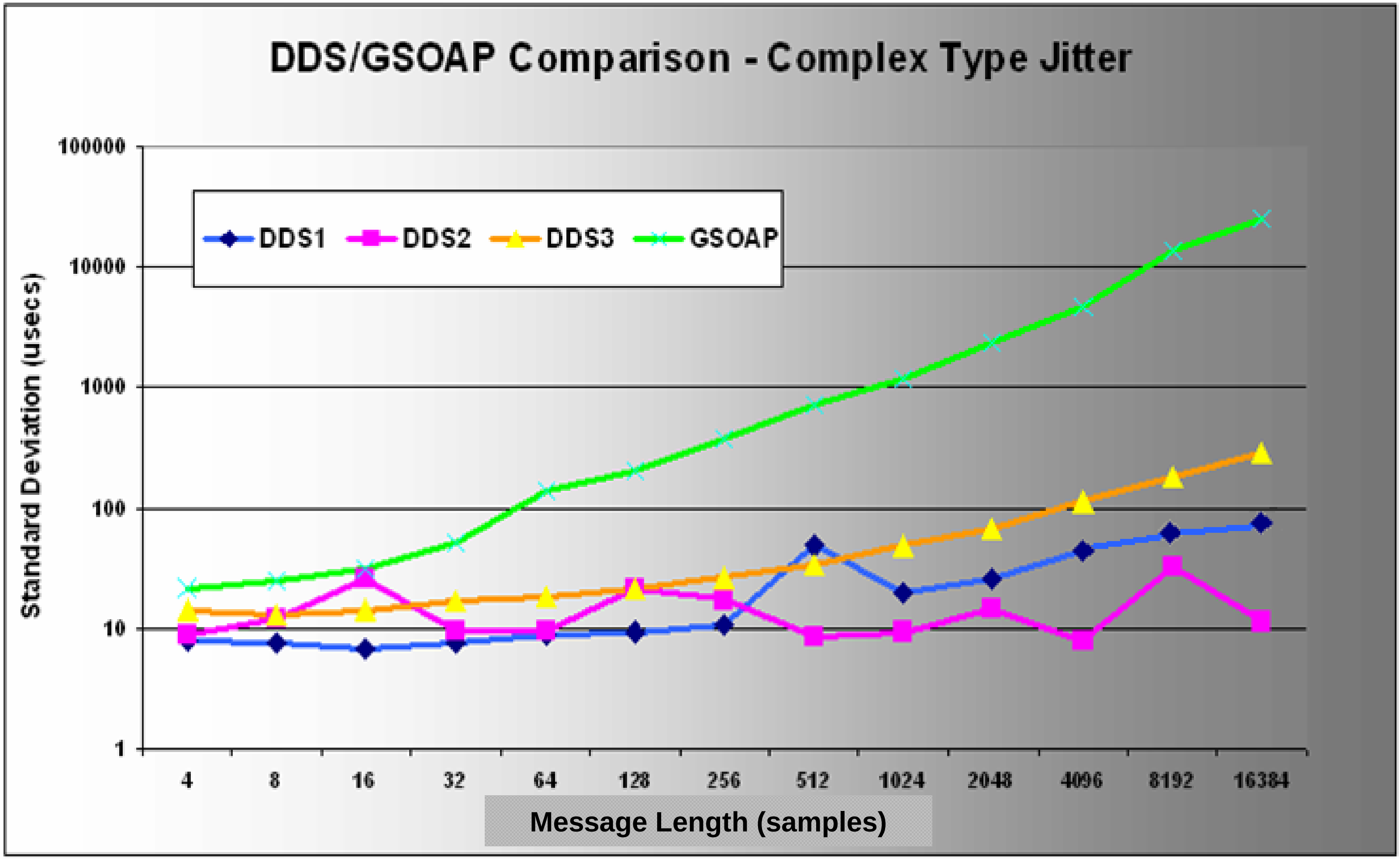
Latency – Complex Data Type



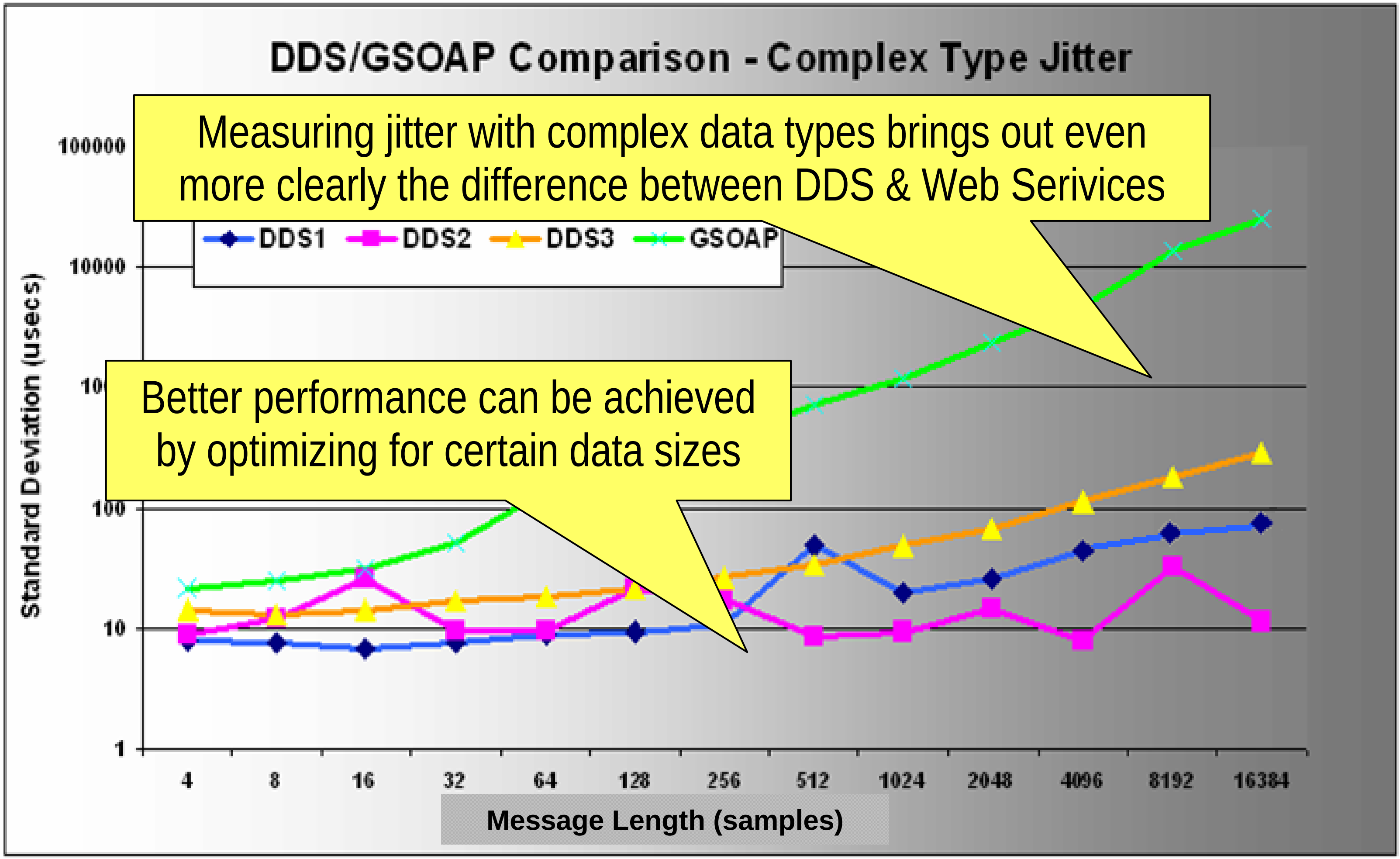
While latency with complex types is less flat for all, DDS still scales better than Web Services by a factor of 2 or more

Some DDS implementations optimized for smaller data sizes

Jitter – Complex Data Type



Jitter – Complex Data Type

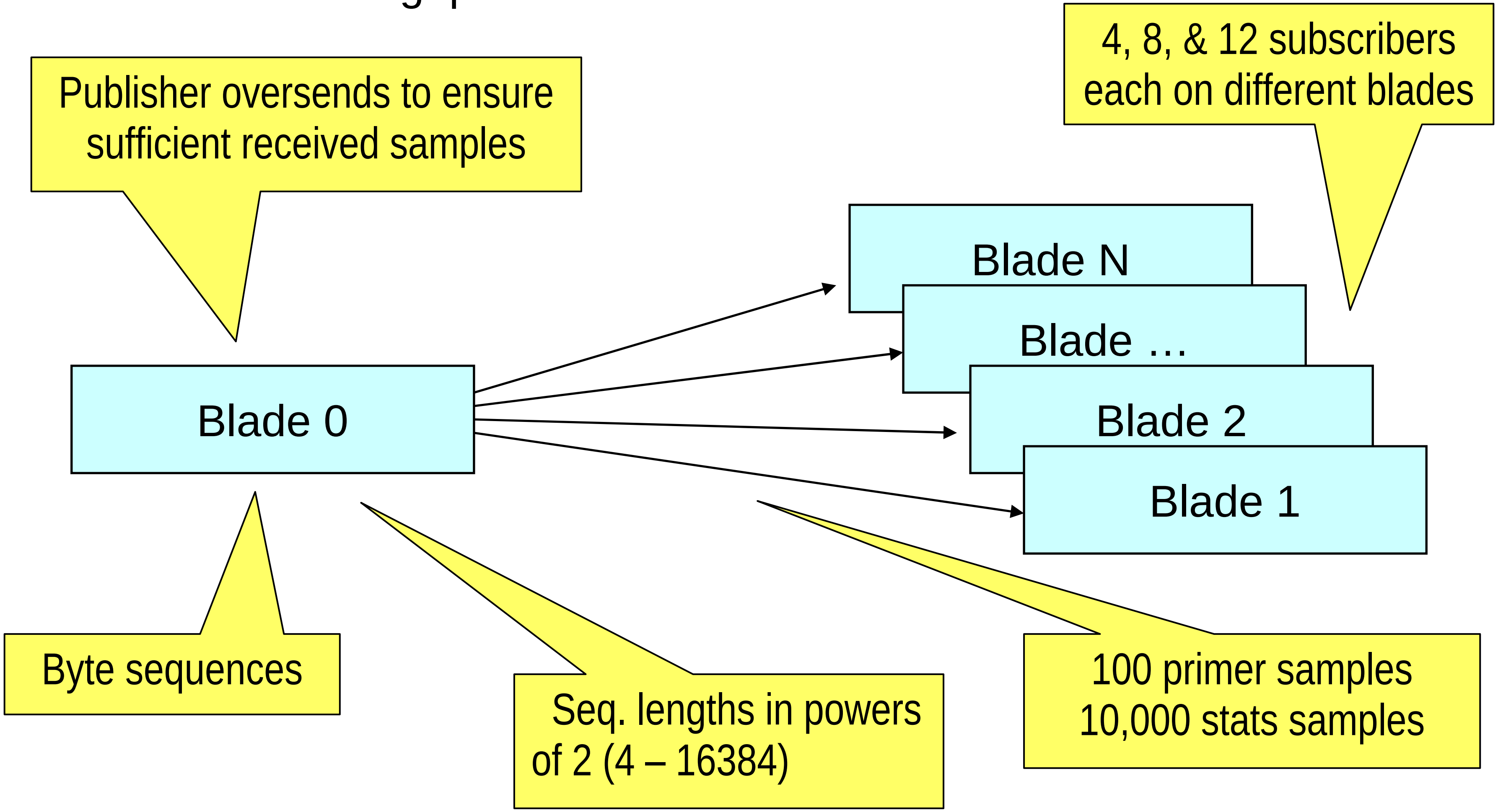


Measuring jitter with complex data types brings out even more clearly the difference between DDS & Web Serivices

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

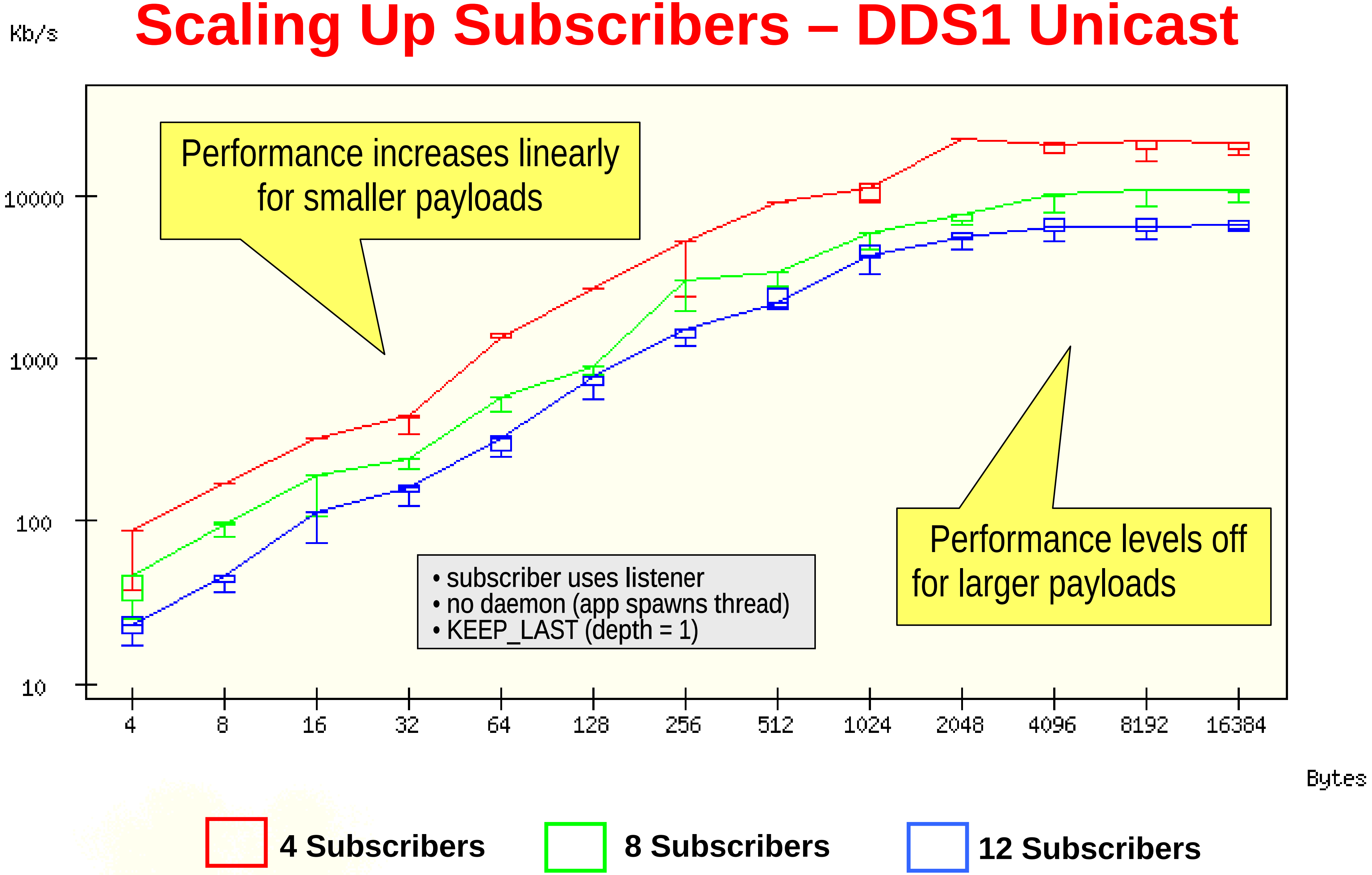
Byte sequences

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



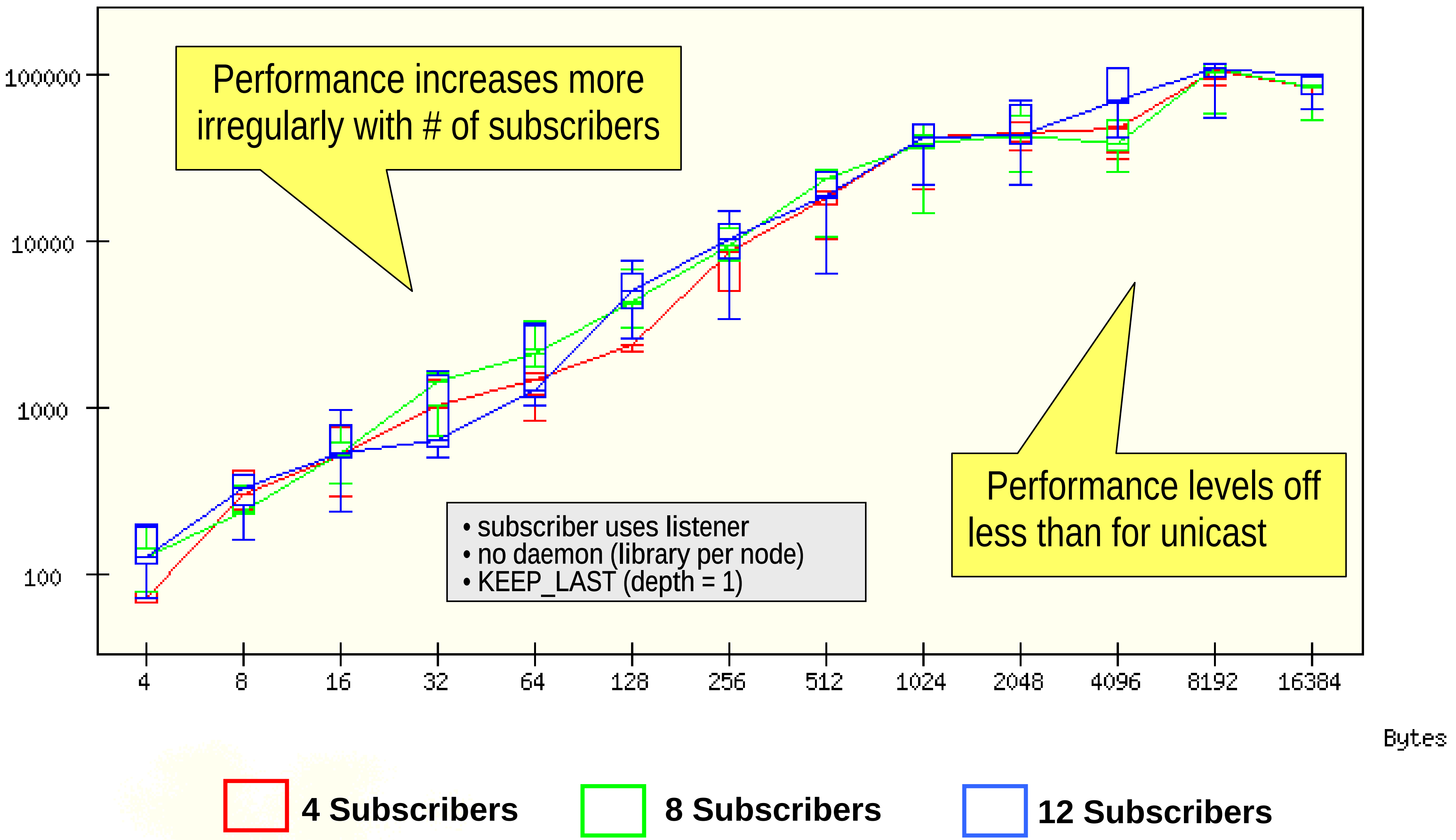
Performance increases linearly for smaller payloads

Performance levels off for larger payloads

- subscriber uses listener
- no daemon (app spawns thread)
- KEEP_LAST (depth = 1)

🟥 4 Subscribers 🟩 8 Subscribers 🟦 12 Subscribers

Scaling Up Subscribers – DDS1 Multicast

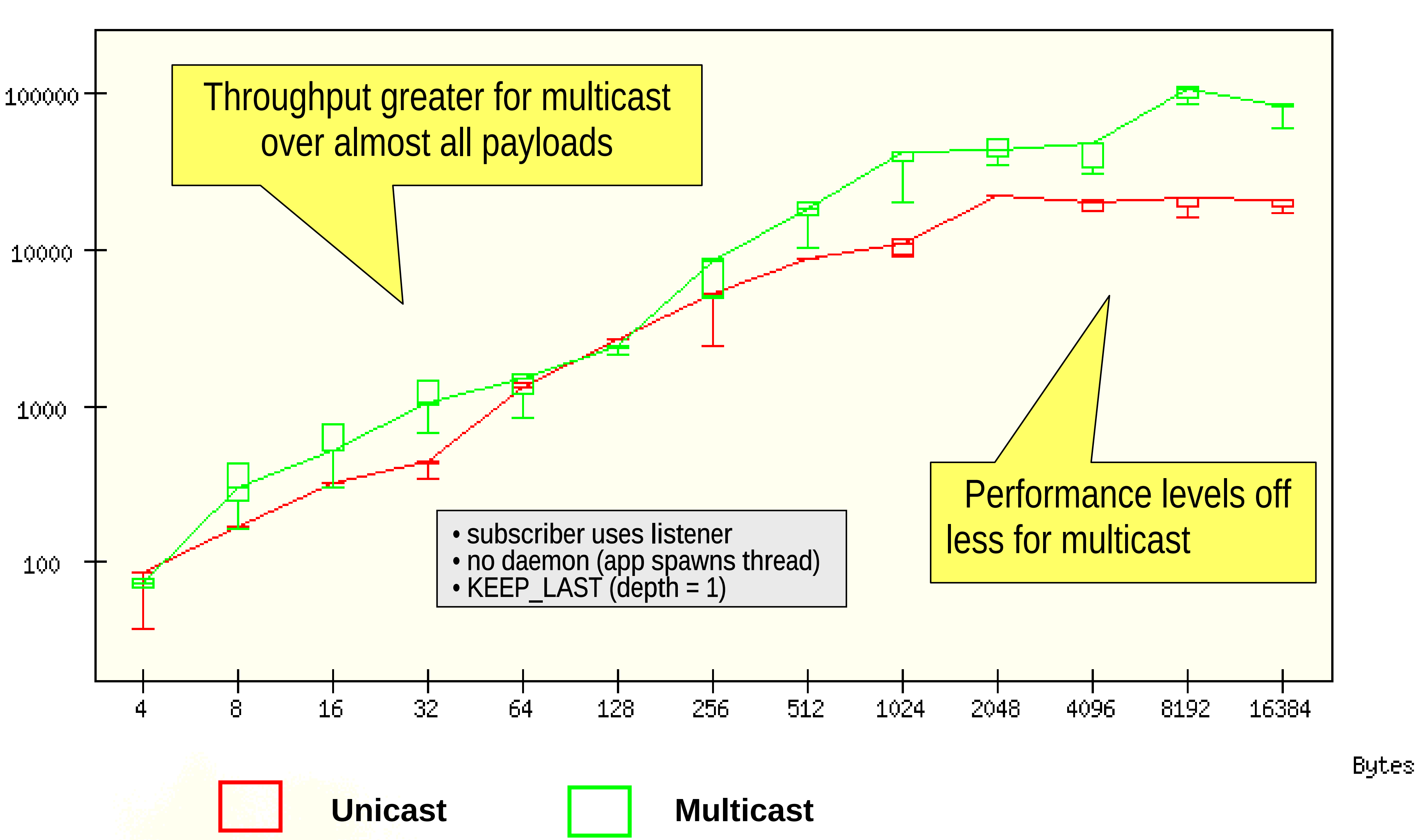


Performance increases more irregularly with # of subscribers

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

Performance levels off less than for unicast

Scaling Up Subscribers – DDS1 1 to 4

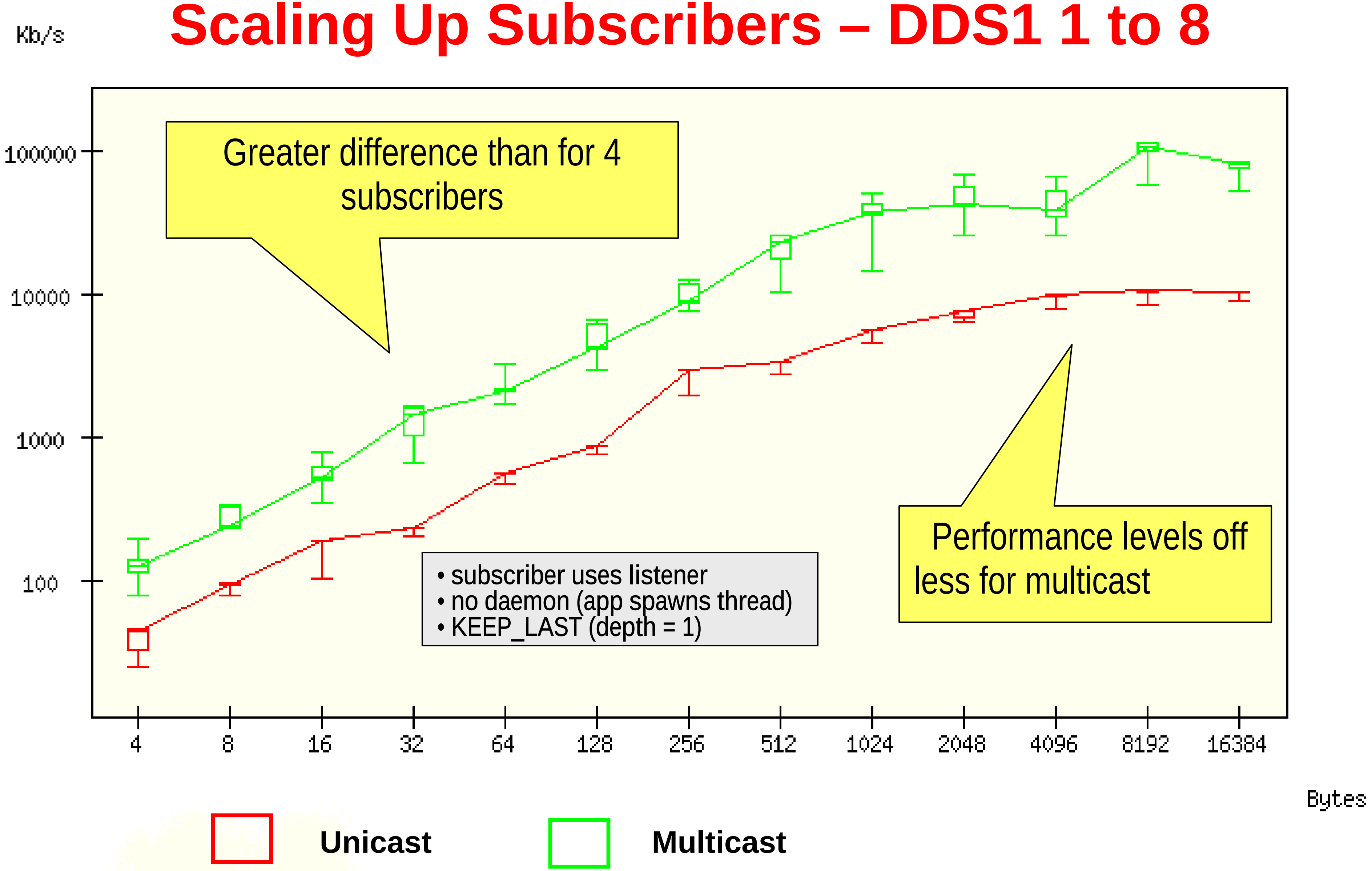


Throughput greater for multicast over almost all payloads

- subscriber uses listener
- no daemon (app spawns thread)
- KEEP_LAST (depth = 1)

Performance levels off less for multicast

Scaling Up Subscribers – DDS1 1 to 8

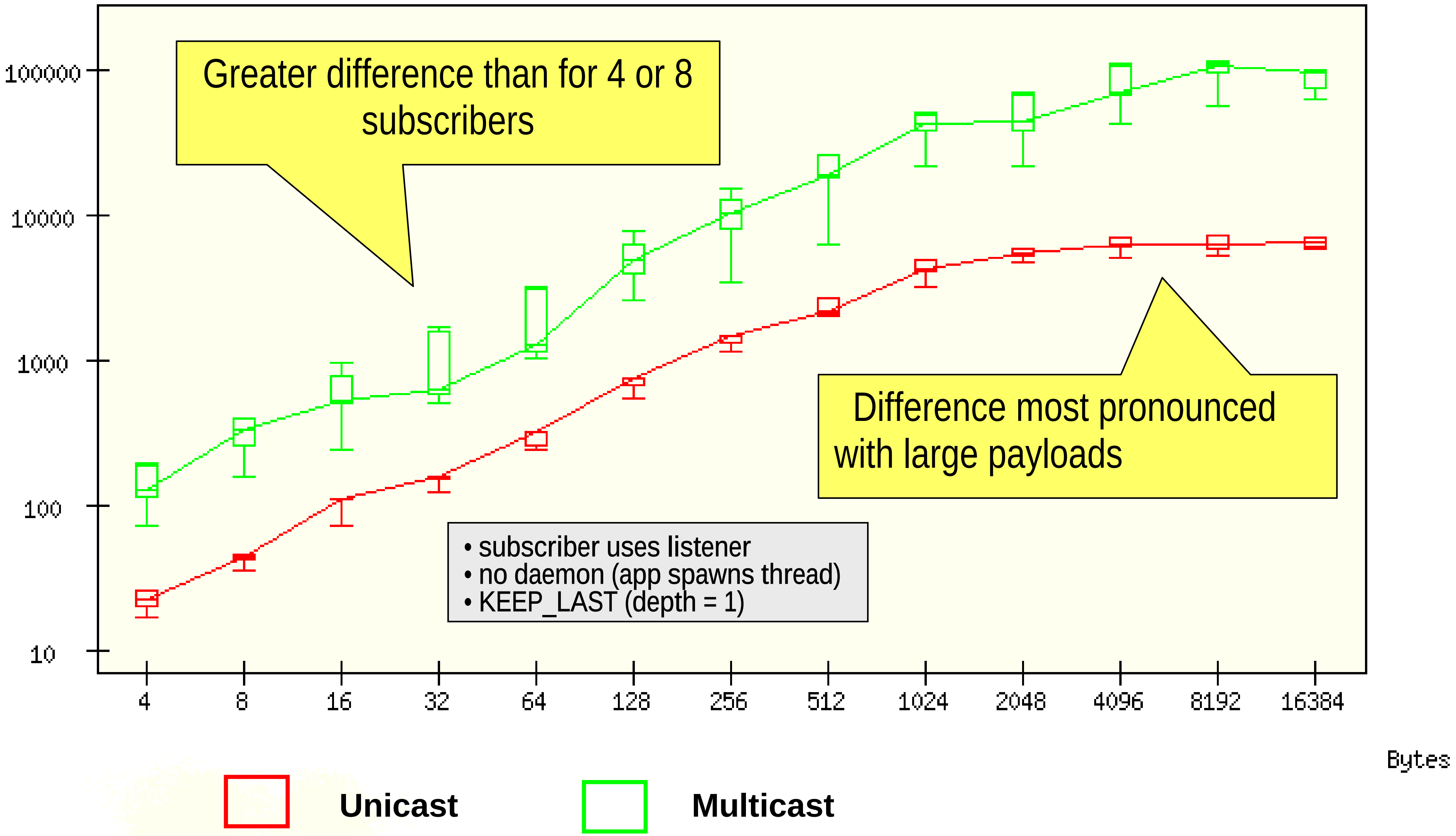


Greater difference than for 4 subscribers

- subscriber uses listener
- no daemon (app spawns thread)
- KEEP_LAST (depth = 1)

Performance levels off less for multicast

Scaling Up Subscribers – DDS1 1 to 12

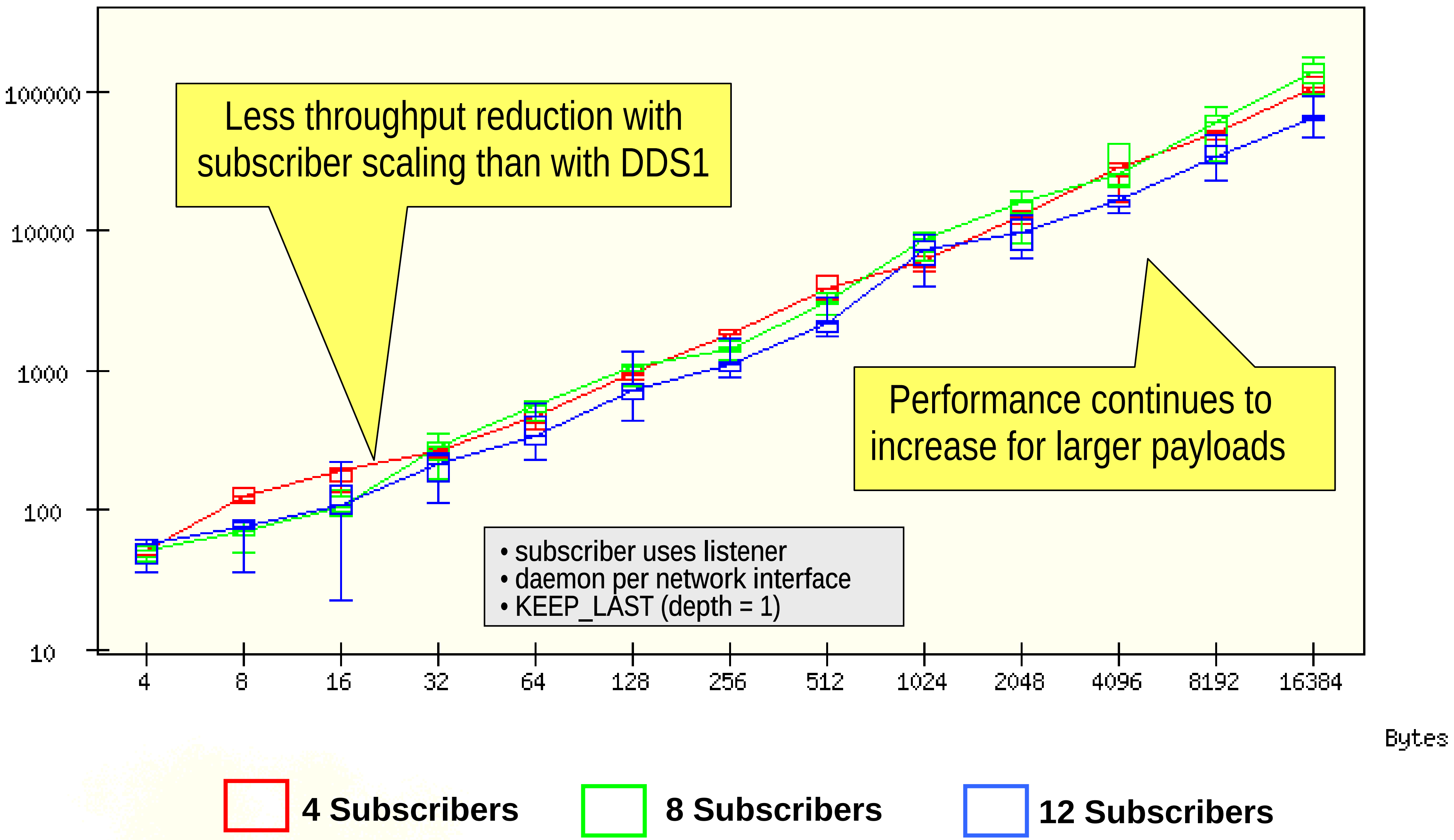


Greater difference than for 4 or 8 subscribers

Difference most pronounced with large payloads

- subscriber uses listener
- no daemon (app spawns thread)
- KEEP_LAST (depth = 1)

Scaling Up Subscribers – DDS2 Broadcast

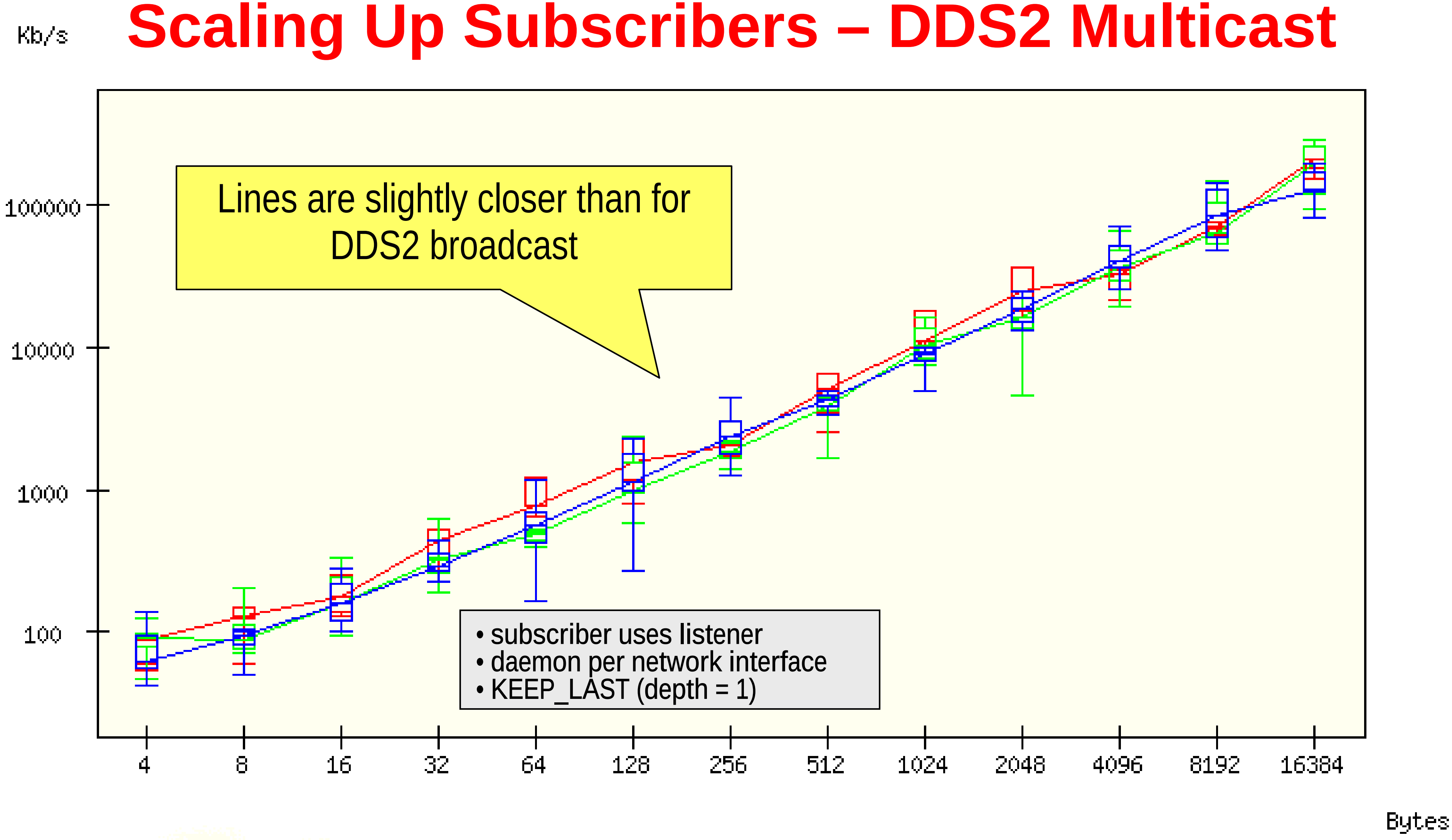


Less throughput reduction with subscriber scaling than with DDS1

Performance continues to increase for larger payloads

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

Scaling Up Subscribers – DDS2 Multicast

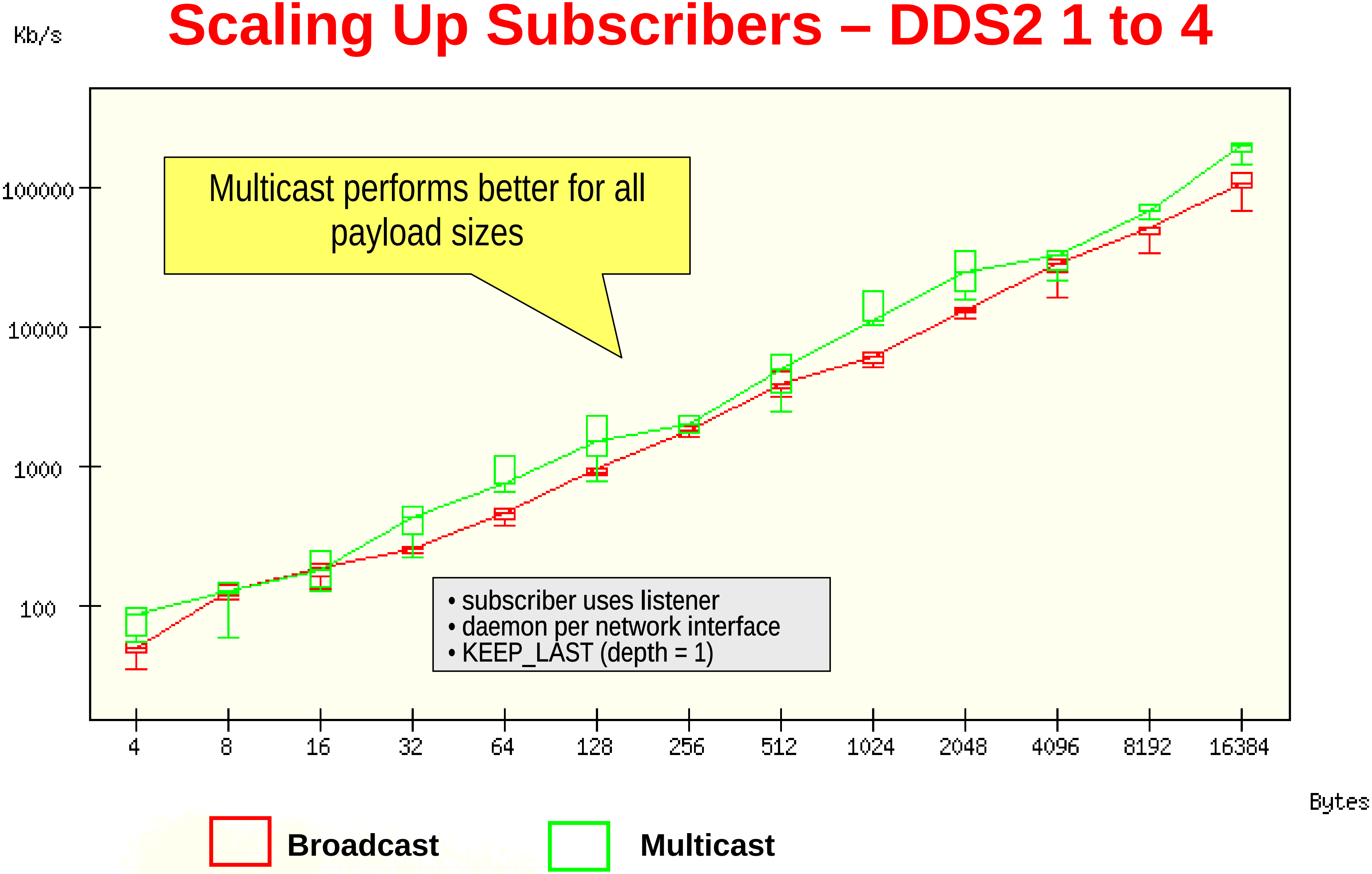


Lines are slightly closer than for DDS2 broadcast

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

🟥 4 Subscribers 🟩 8 Subscribers 🟦 12 Subscribers

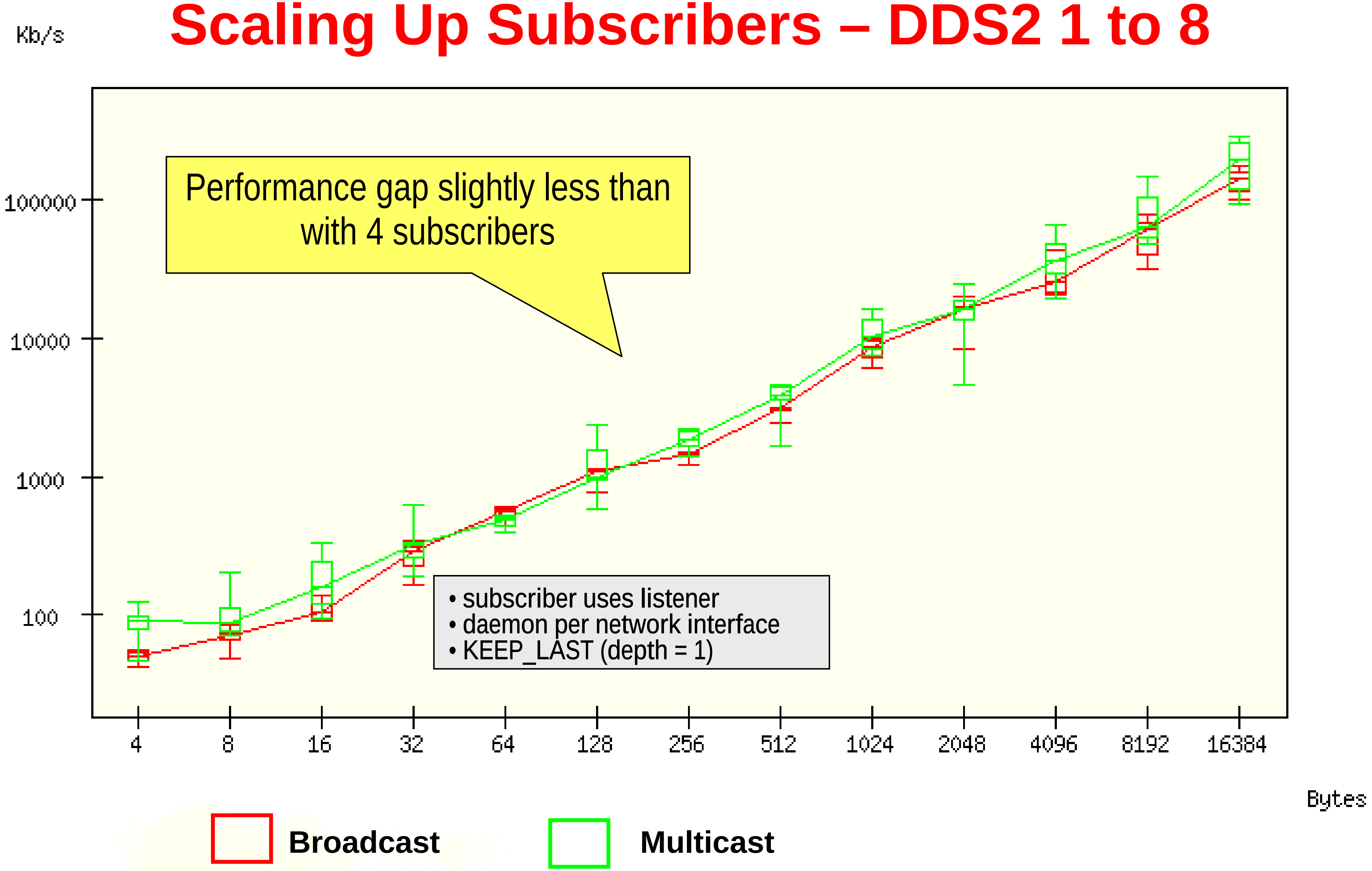
Scaling Up Subscribers – DDS2 1 to 4



Multicast performs better for all payload sizes

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

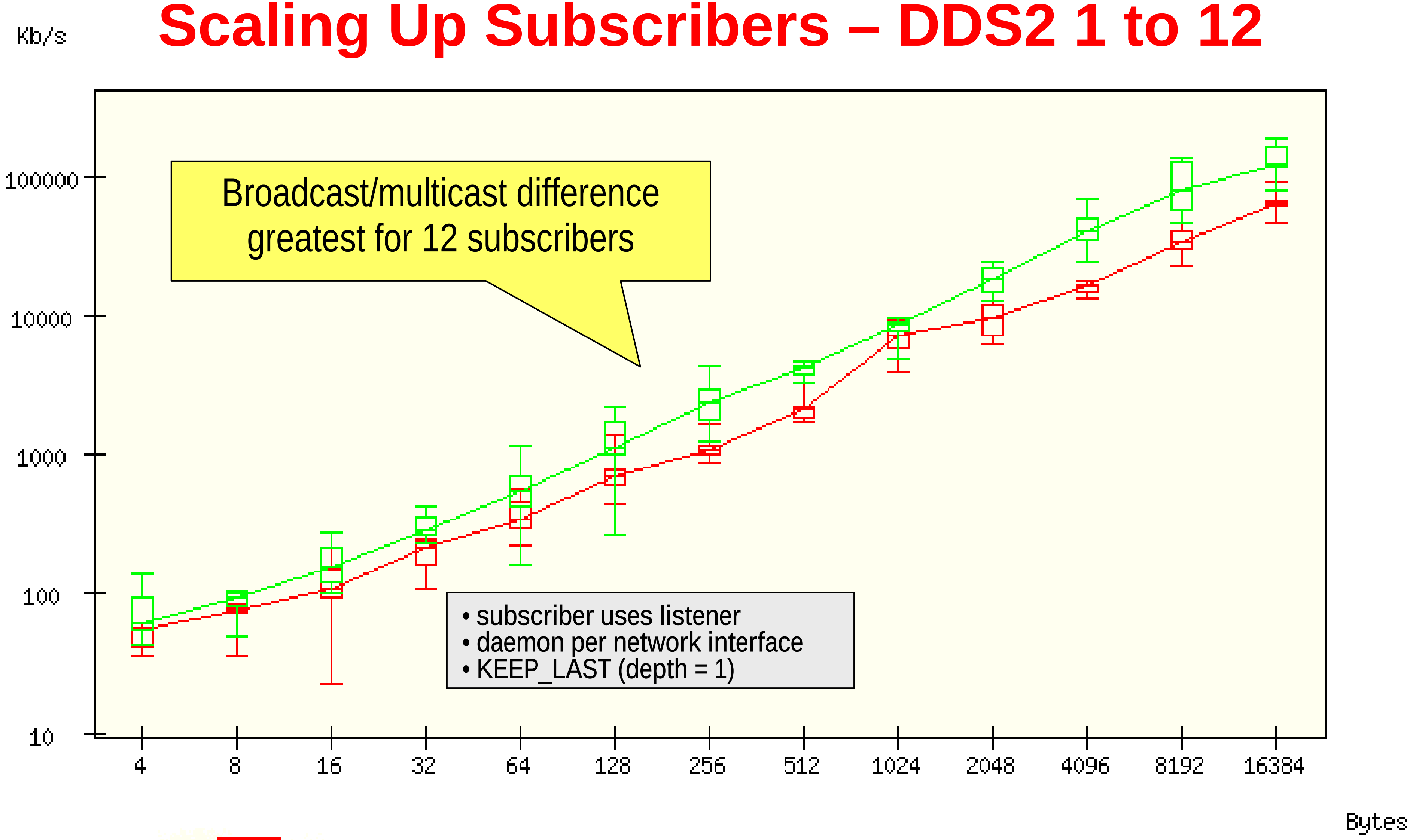
Scaling Up Subscribers – DDS2 1 to 8



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

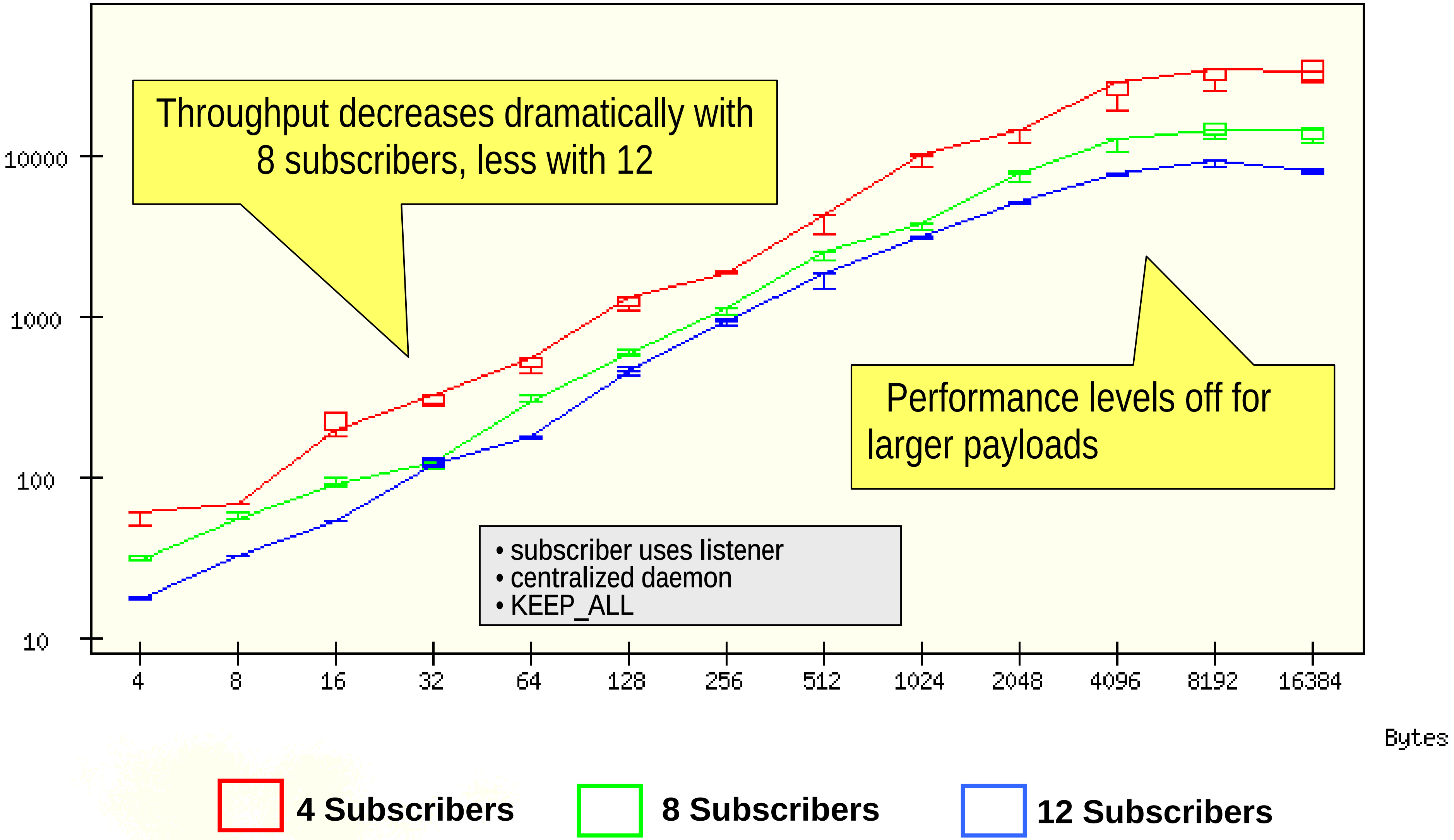


Broadcast/multicast difference greatest for 12 subscribers

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

🟥 **Broadcast** 🟩 **Multicast**

Scaling Up Subscribers – DDS3 Unicast

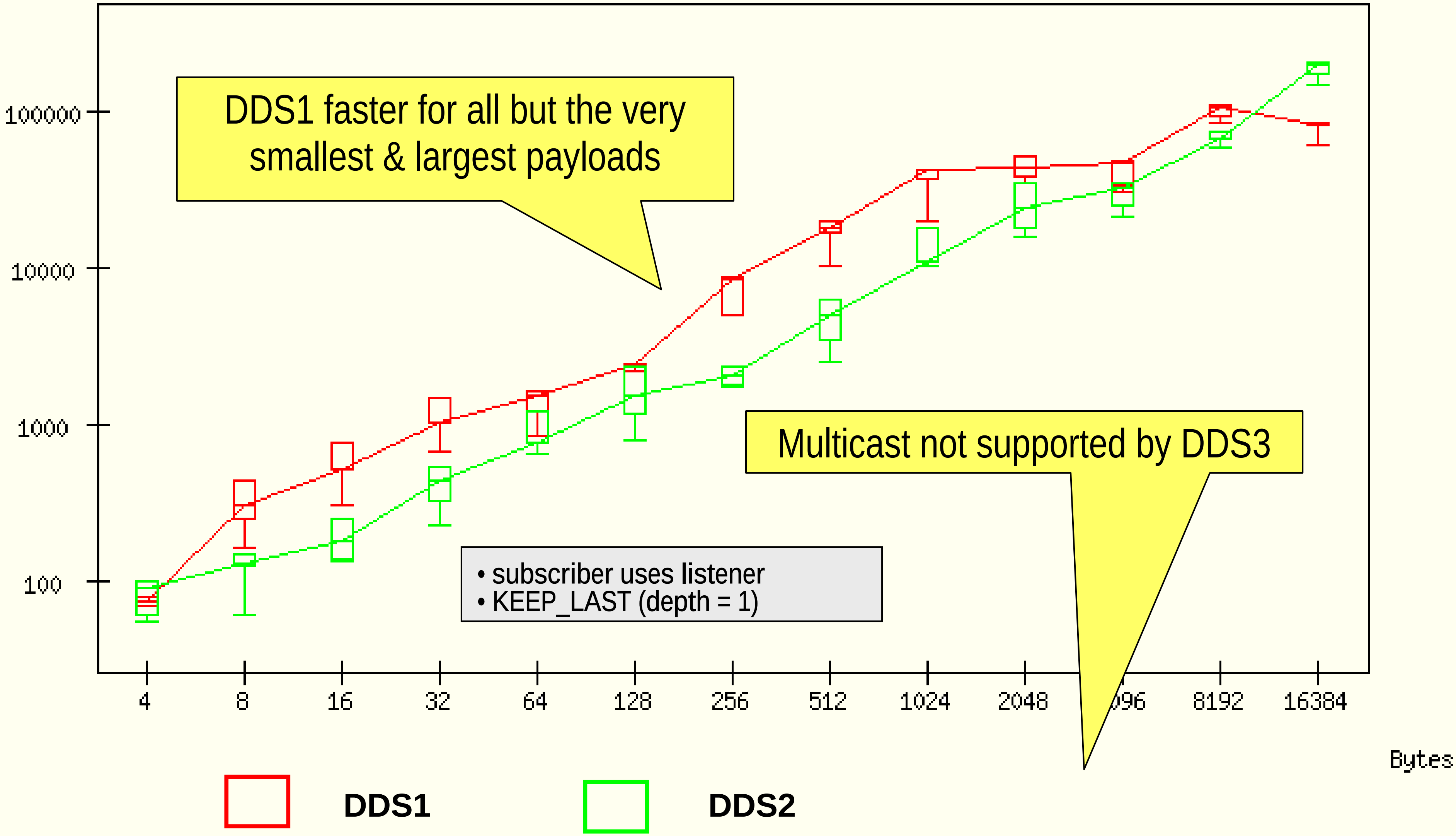


Throughput decreases dramatically with 8 subscribers, less with 12

Performance levels off for larger payloads

- subscriber uses listener
- centralized daemon
- KEEP_ALL

Impl Comparison: 4 Subscribers Multicast



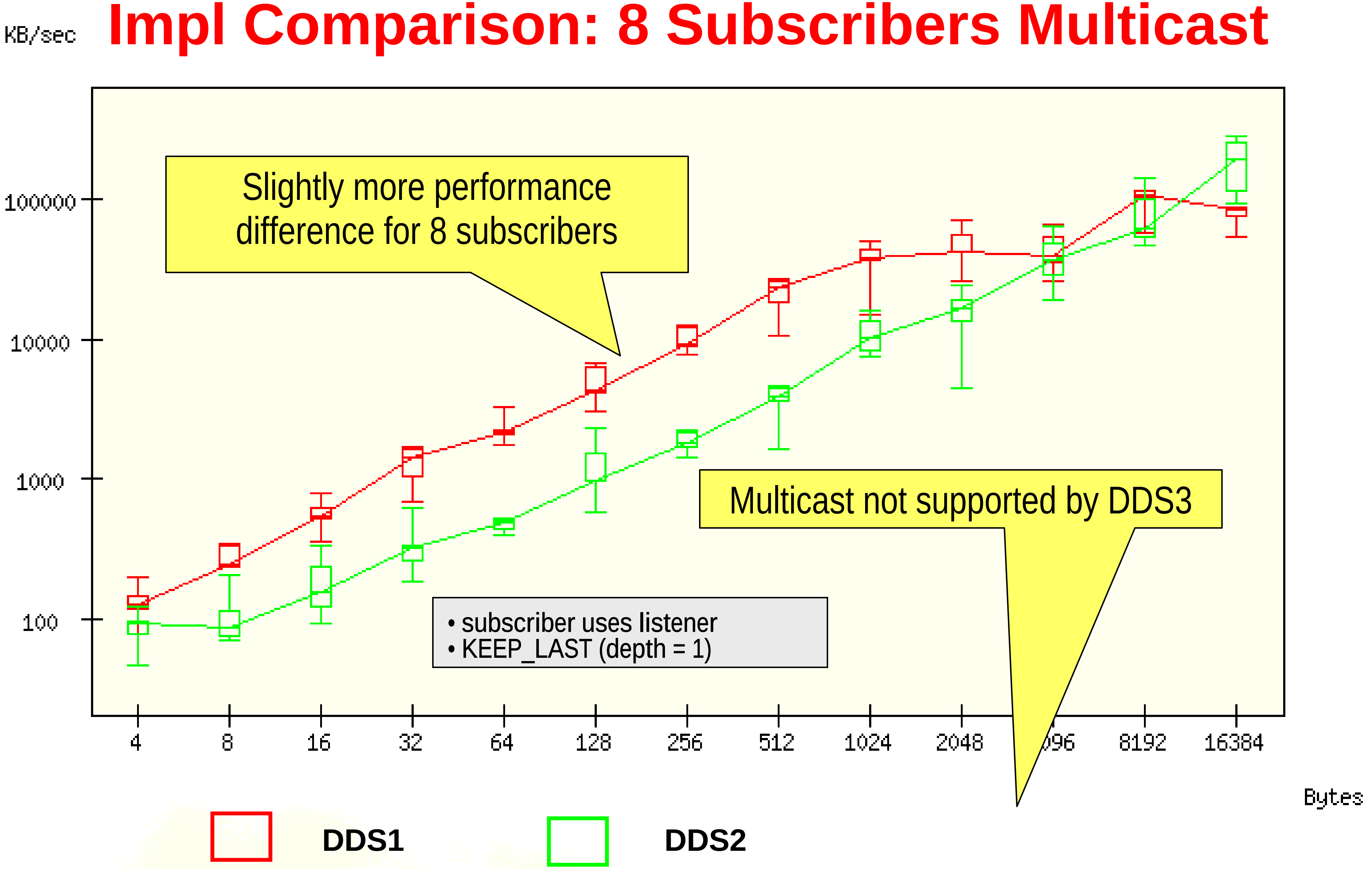
DDS1 faster for all but the very smallest & largest payloads

Multicast not supported by DDS3

- subscriber uses listener
- KEEP_LAST (depth = 1)

🟥 DDS1 🟩 DDS2

Impl Comparison: 8 Subscribers Multicast

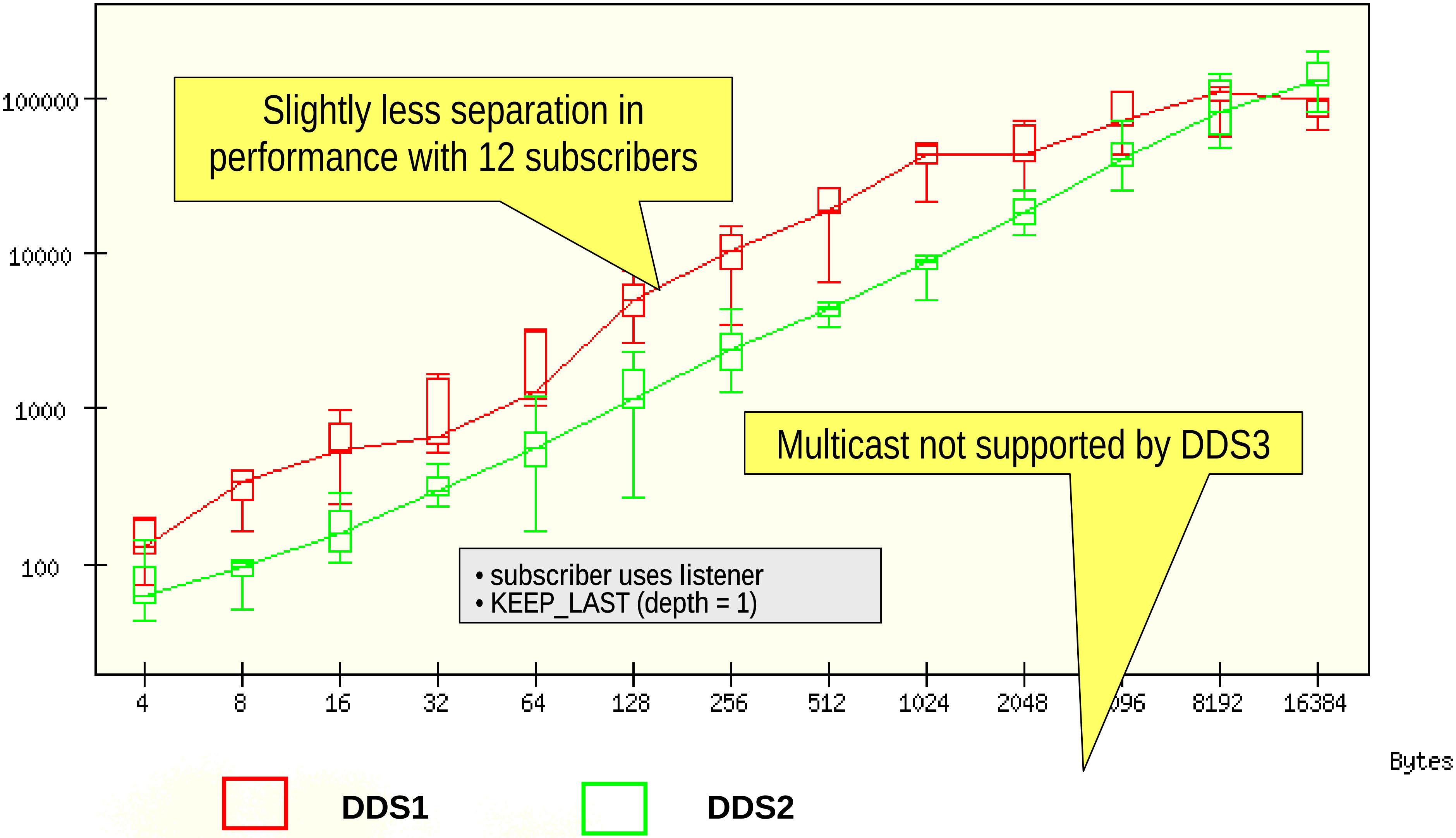


Slightly more performance difference for 8 subscribers

Multicast not supported by DDS3

- subscriber uses listener
- KEEP_LAST (depth = 1)

Impl Comparison: 12 Subscribers Multicast

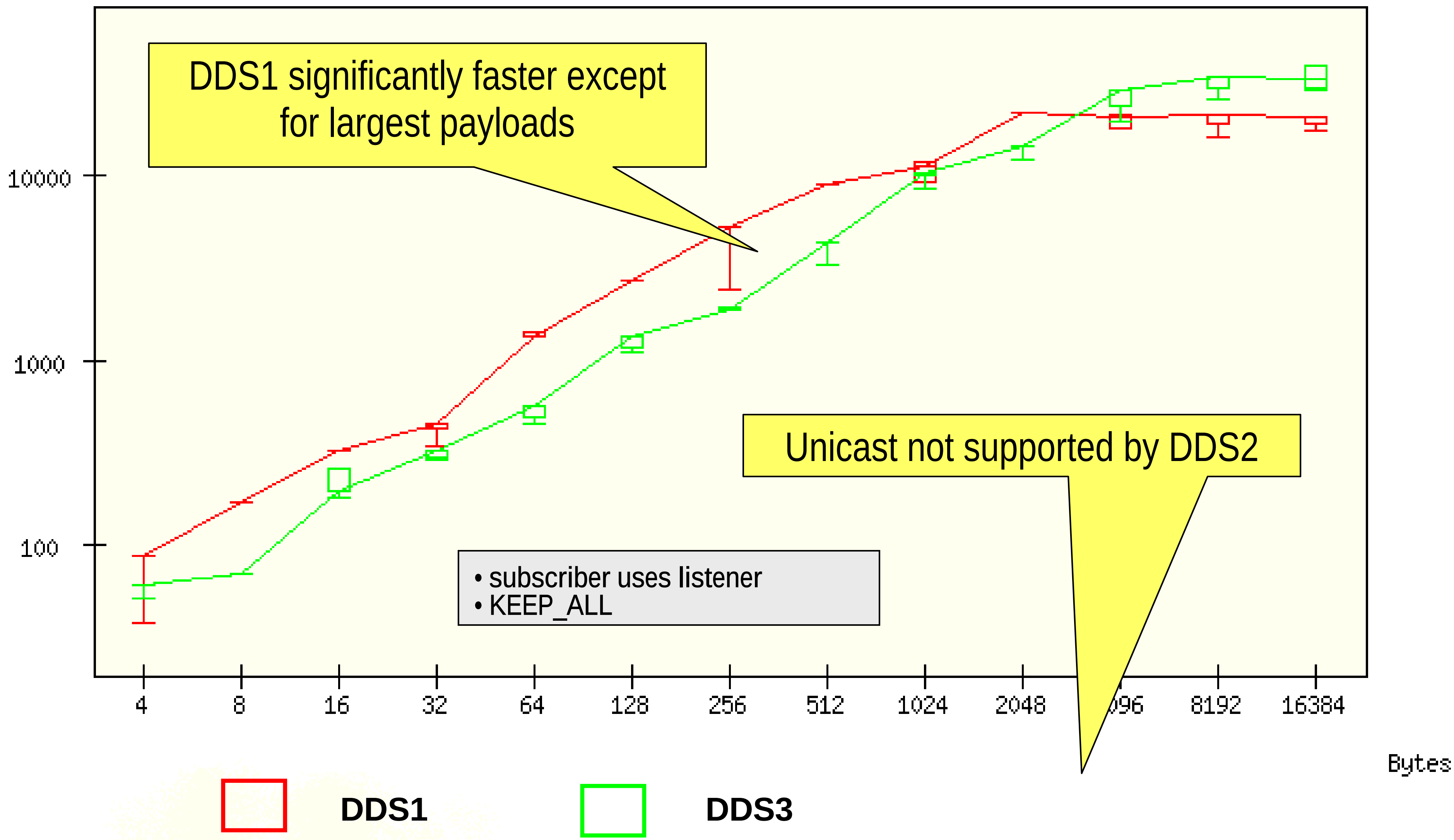


Slightly less separation in performance with 12 subscribers

Multicast not supported by DDS3

- subscriber uses listener
- KEEP_LAST (depth = 1)

Impl Comparison: 4 Subscribers Unicast

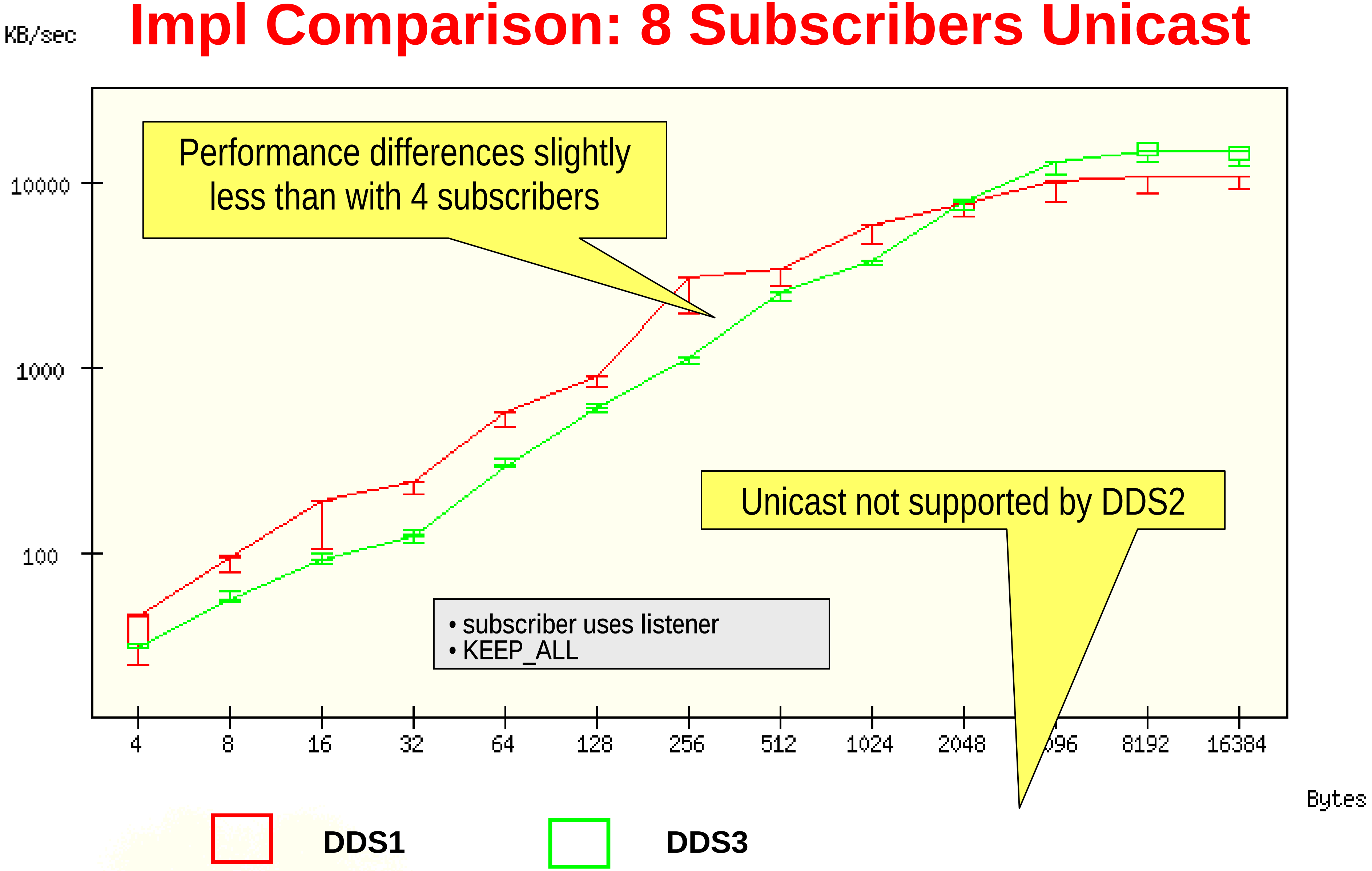


DDS1 significantly faster except for largest payloads

Unicast not supported by DDS2

- subscriber uses listener
- KEEP_ALL

Impl Comparison: 8 Subscribers Unicast

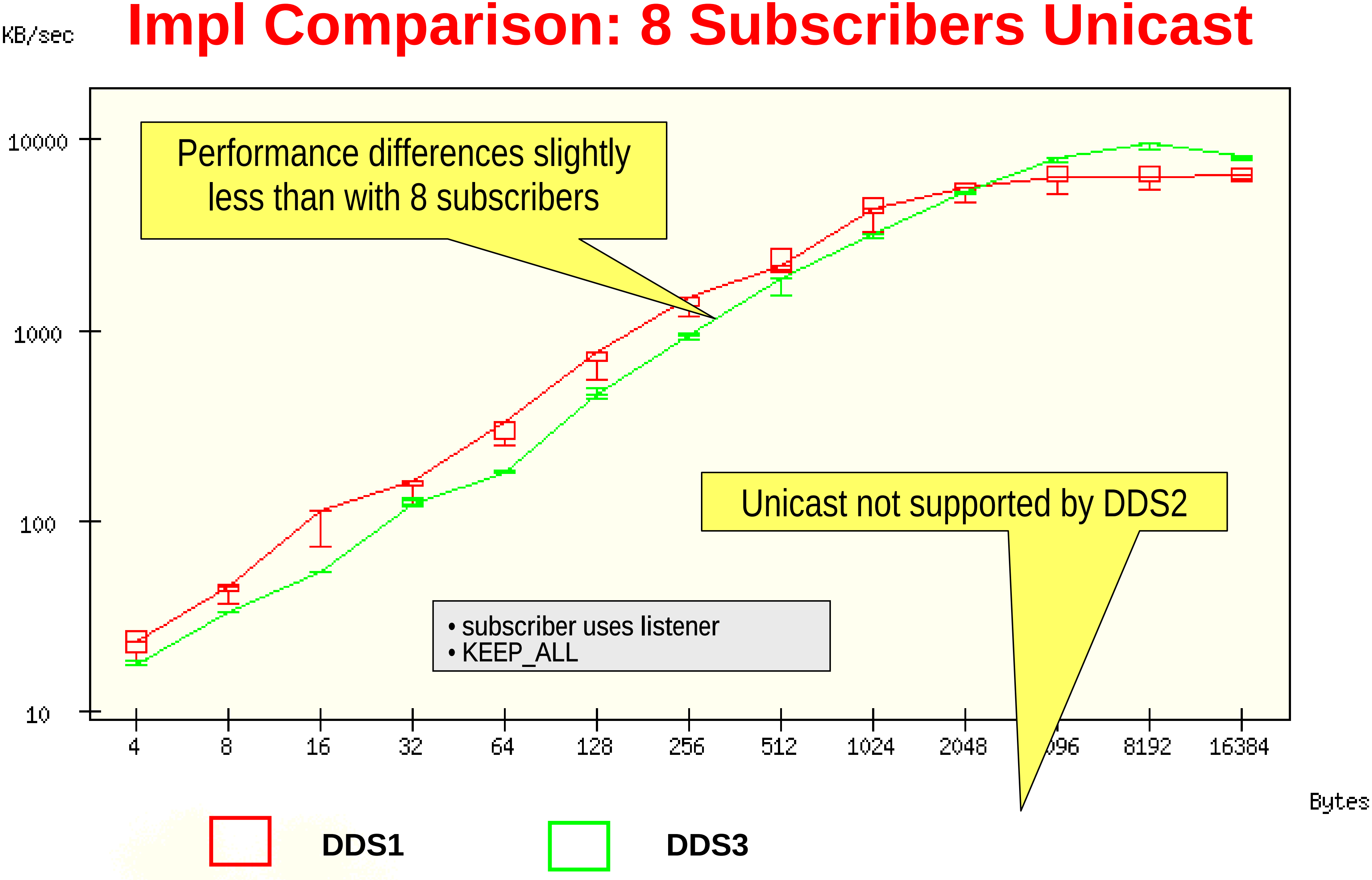


Performance differences slightly less than with 4 subscribers

Unicast not supported by DDS2

- subscriber uses listener
- KEEP_ALL

Impl Comparison: 8 Subscribers Unicast

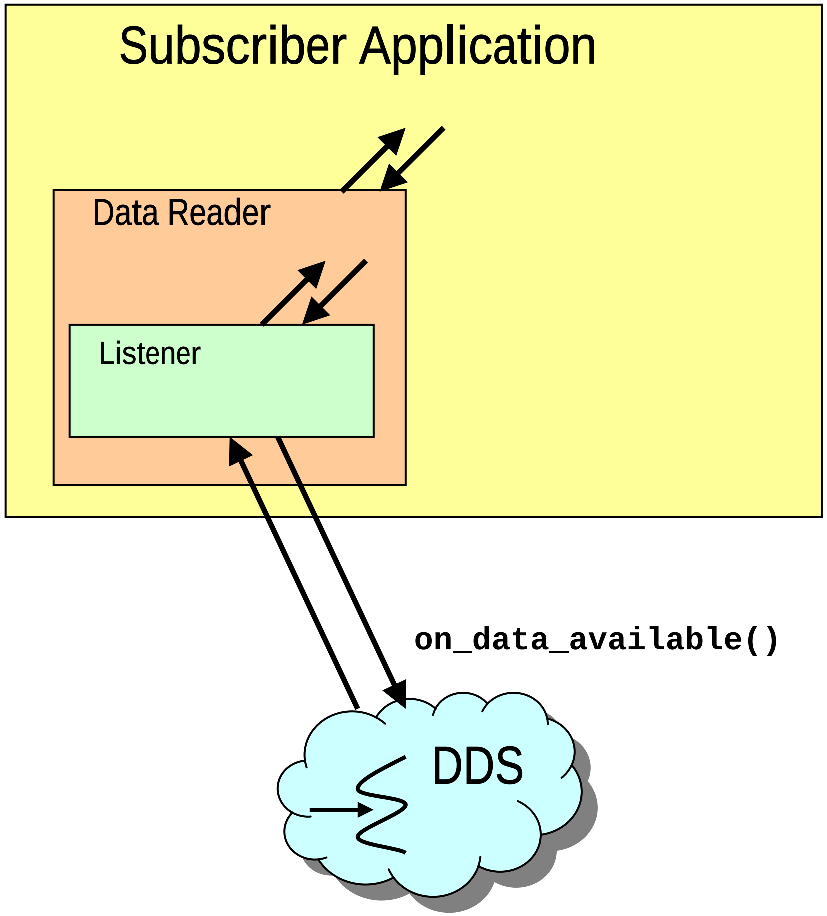


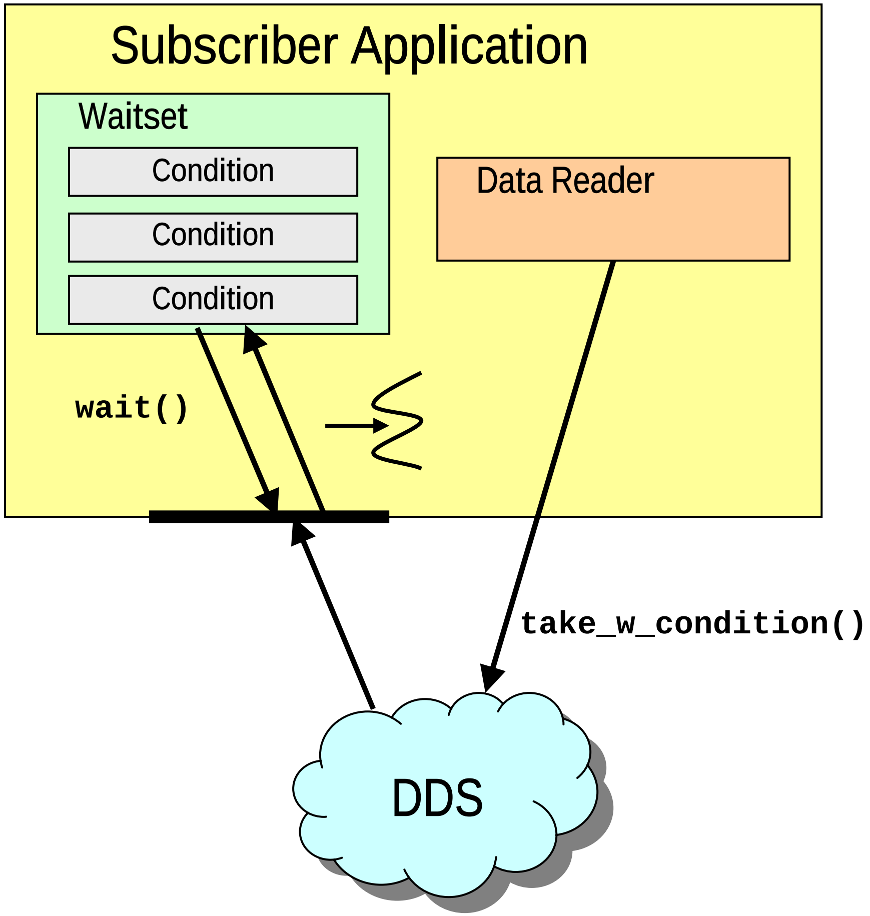
Performance differences slightly less than with 8 subscribers

Unicast not supported by DDS2

- subscriber uses listener
- KEEP_ALL

Overview of DDS Listener vs. Waitset

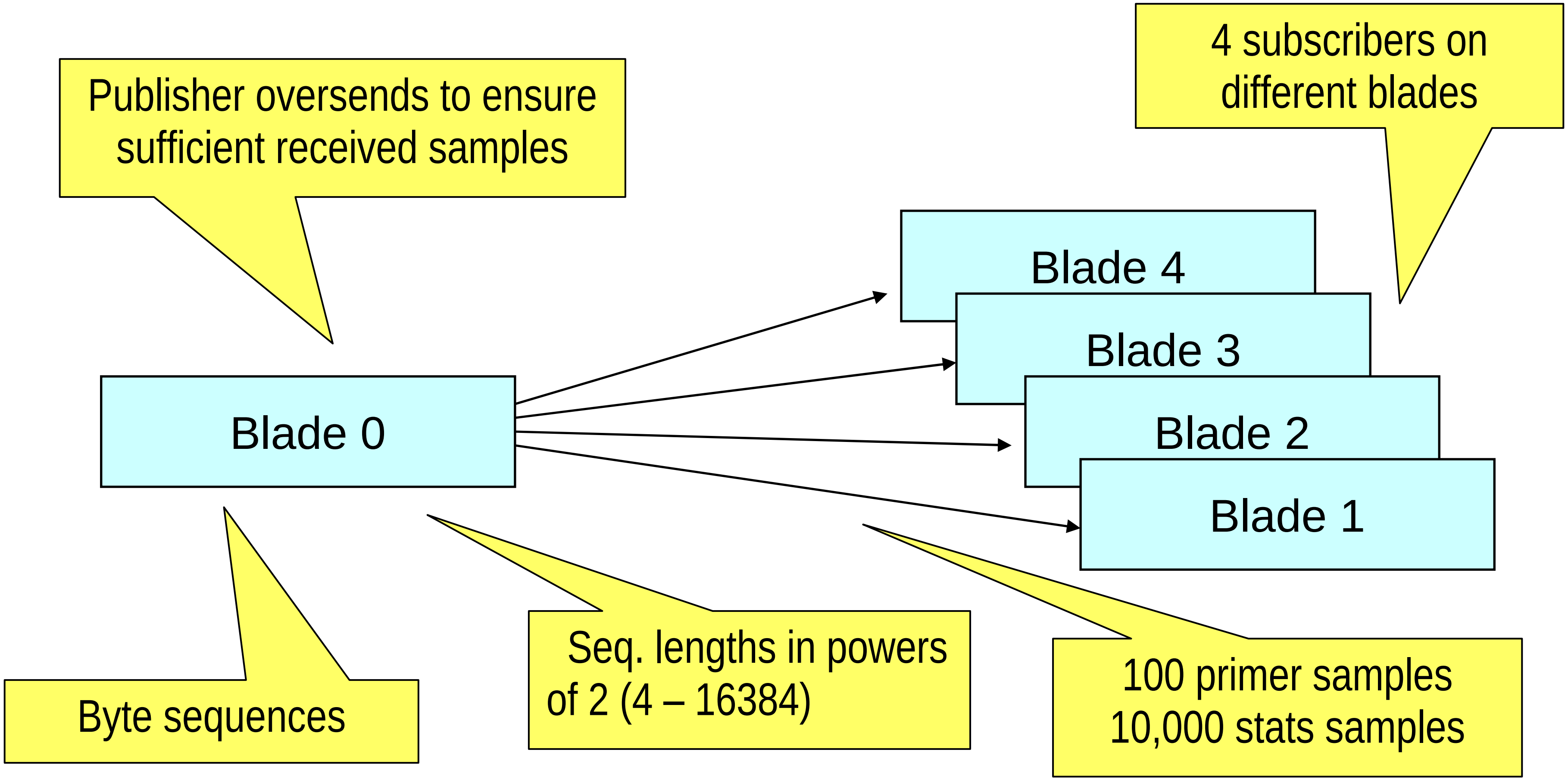




- No blocking
- DDS thread executes unknown application code

- 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 3

Blade 2

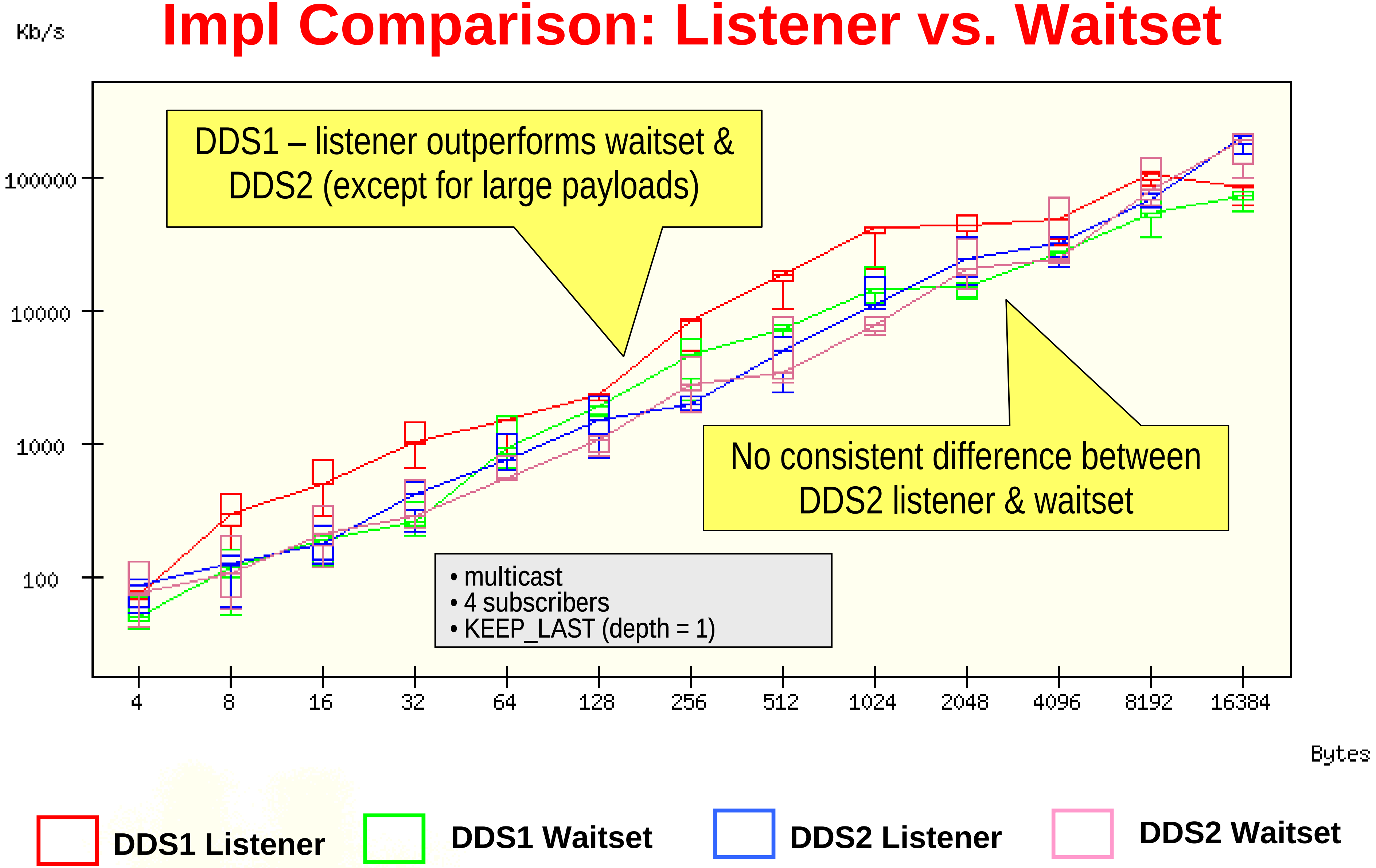
Blade 1

Byte sequences

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

100 primer samples
10,000 stats samples

Impl Comparison: Listener vs. Waitset



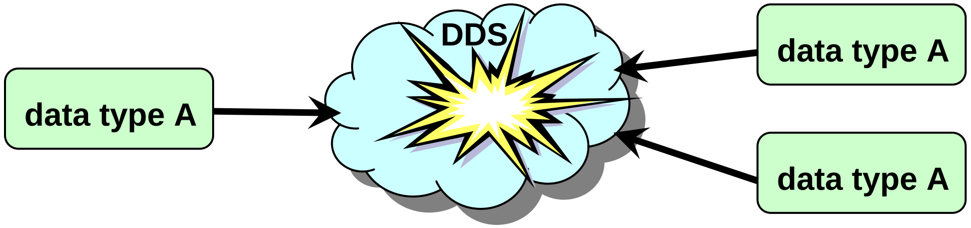
DDS1 – listener outperforms waitset & DDS2 (except for large payloads)

No consistent difference between DDS2 listener & waitset

- multicast
- 4 subscribers
- KEEP_LAST (depth = 1)

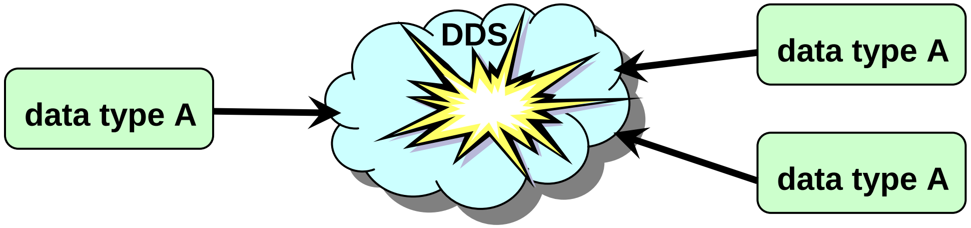
DDS Application Challenges

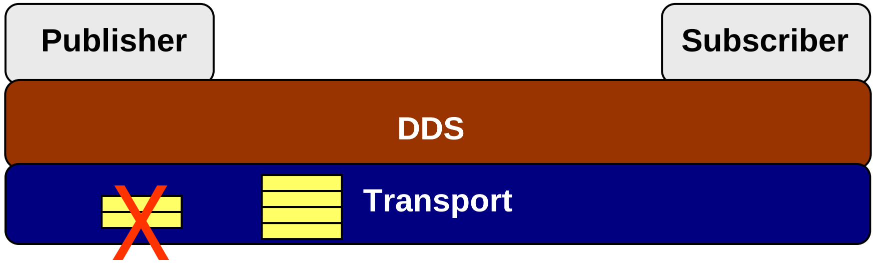
- Scaling up number of subscribers
  - Data type registration race condition (DDS3)
  - Setting proprietary 'participant index' QoS (DDS1)



DDS Application Challenges

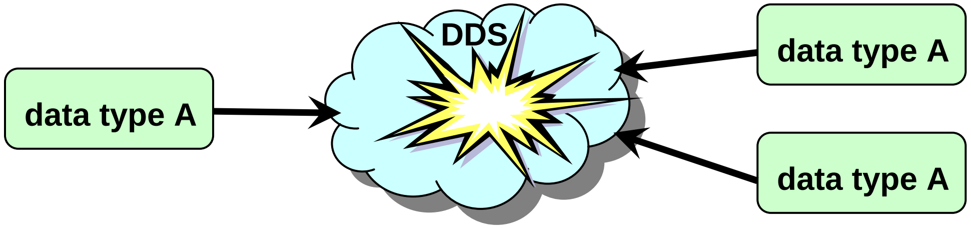
- Scaling up number of subscribers
  - Data type registration race condition (DDS3)
  - Setting proprietary ‘participant index’ QoS (DDS1)
- Getting a sufficient transport buffer size

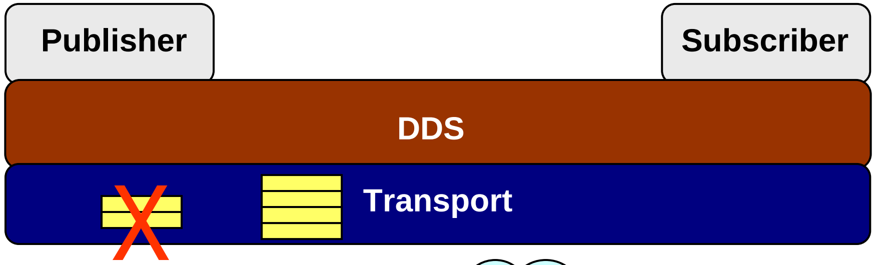


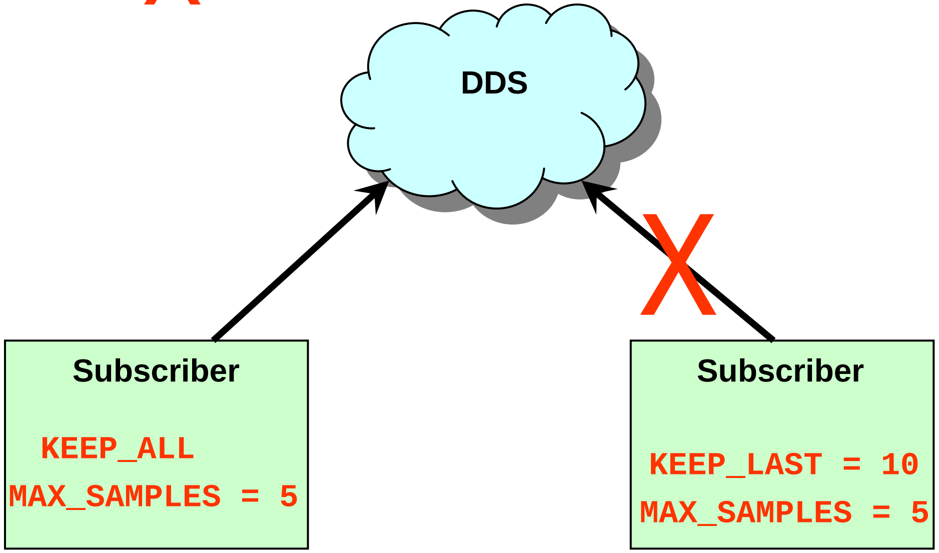


DDS Application Challenges

- 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

	DDS1	DDS2	DDS3
DomainParticipant Factory	compliant	compliant	proprietary function
Register Data Types	static method	member method	member method
Spec Operations	extra argument (newer spec)	compliant	compliant
Key Declaration	`//@key`	single `#pragma`	pair of `#pragma`
Required App. IDs	publisher & subscriber	none	publisher
Required App. Transport Config	code-based	none	file-based or code-based

Portability Challenges

	DDS1	DDS2	DDS3
DomainParticipant Factory	compliant	compliant	proprietary function
Register Data Types	[illegible]	member method	member method
Spec Operations	extra argument (newer spec)	compliant	compliant
Key Declaration	[illegible]	single pragma	pair of pragma
Required App. IDs	publisher & subscriber	none	publisher
Required App. Transport Config	code-based	none	file-based or code-based

```
DomainParticipantFactory::get_instance();
```

```
TheParticipantFactoryWithArgs(argc, argv);
```

Portability Challenges

	DDS1	DDS2	DDS3
DomainParticipant Factory	compliant	compliant	proprietary function
Register Data Types	static method	member method	member method
	extra ar	t	compliant
Key Declaration	//@key	single #pragm	pair of ragma
Required App. IDs	subscriber	none	publisher
Required App. Transport Config	code-based	none	file-based or code-based

```
DataType::register_type(participant, name);
```

```
DataType identifier;
identifier.register_type(participant, name);
```

Portability Challenges

	DDS1	DDS2	DDS3
DomainParticipant Factory	compliant	compliant	proprietary function
Register Data Types	static method	member method	member method
Spec Operations	extra argument (newer spec)	compliant	compliant
Key Declaration	//	single	ir of
R	pu	none	publisher
Transport Config		none	file-based or code-based

```
create_publisher(QoS_list,
                 listener,
                 DDS_StatusKind);
```

```
create_publisher(QoS_list,
                 listener);
```

Portability Challenges

	DDS1	DDS2	DDS3
Dor Factory	nt	compliant	proprietary function
Register Data Types	stati thod	member method	member method
Spec Operations	extra argum t (newer spec)	compliant	compliant
Key Declaration	//@key	single #pragma	pair of #pragma
	publisher & subs		lisher
	code-based	none	file-based or code-based

```
#pragma keylist Info id
```

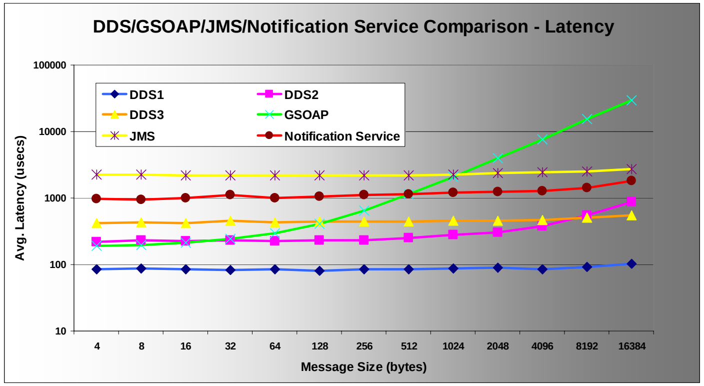
```
struct Info {
  long id; //@key
  string msg;
};
```

```
#pragma DCPS_DATA_TYPE “Info”
#pragma DCPS_DATA_KEY “id”
```

Lessons Learned

Pros

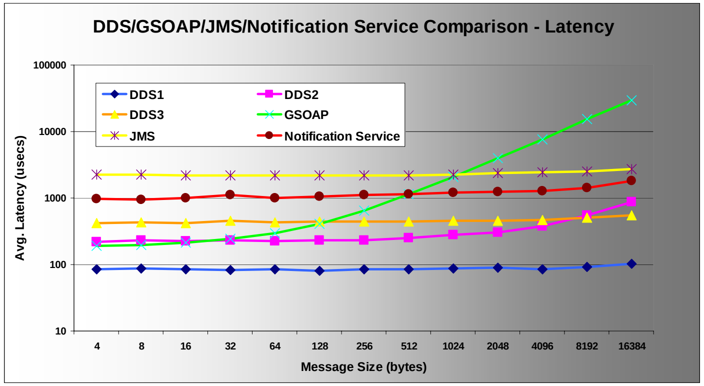
- Performance of DDS is significantly faster than other pub/sub architectures
  - Even the slowest was 2x faster than other pub/sub services
- DDS scales better to larger payloads, especially for simple data types

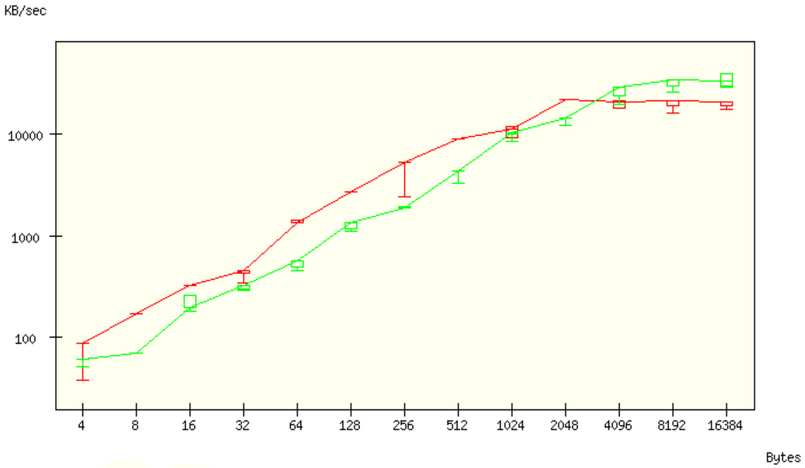


Lessons Learned

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

Impl	unicast	multicast	broadcast
DDS1	**Yes (default)**	**Yes**	**No**
DDS2	**No**	**Yes**	**Yes (default)**
DDS3	**Yes (default)**	**No**	**No**

  - 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
- 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

- Investigate migration of processing to source
- Measure discovery time
- Include Java DDS implementations
- Include other pub/sub platforms
  - WS Notification
  - ICE pub/sub
- Find scenarios that distinguish performance of QoS policies & features, e.g.:
  - Listener vs waitset
  - Collocated applications
  - Very large # of subscribers & payload sizes

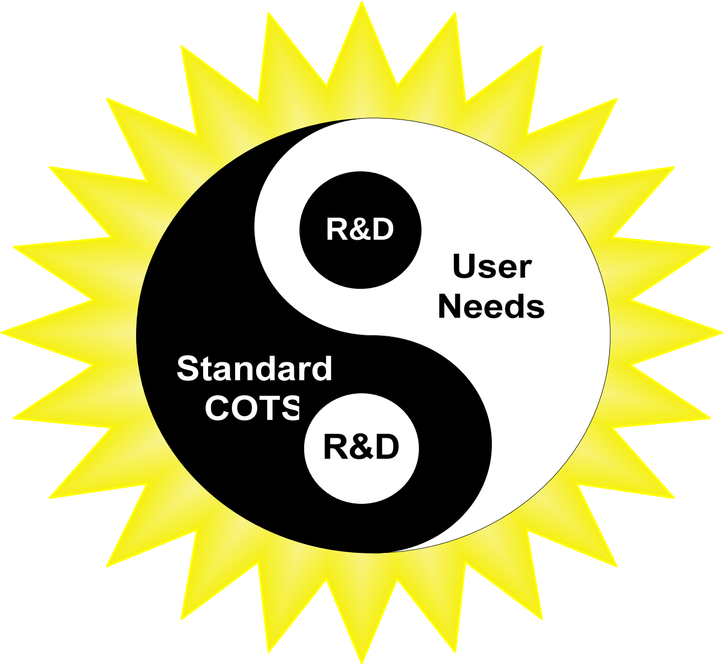
Future Work - Benchmarking Framework

- 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

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 work at www.dre.vanderbilt.edu/DDS
- Latest version of these slides at DDS_RTWS06.pdf in the above directory



- Thanks to Object Computing Inc (OCI), PrismTechnologies (PrismTech), & Real-time Innovations (RTI) for providing their DDS implementations & for helping with the benchmarking process
- Thanks to AFRL/IF, Vanderbilt, & NSF for their sponsorship