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

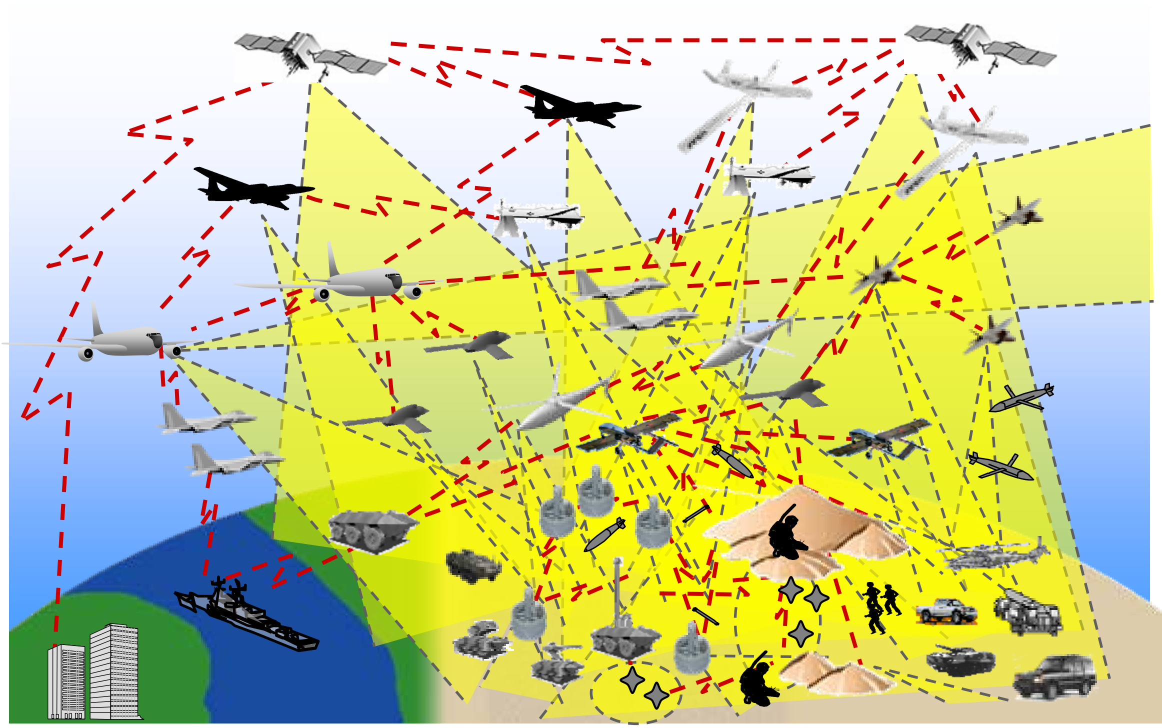
Demands on Tactical Information Systems

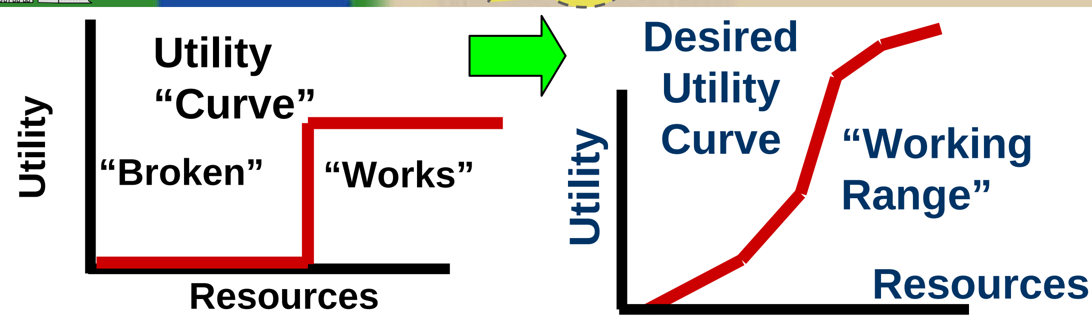
Key ***problem space*** challenges

- 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

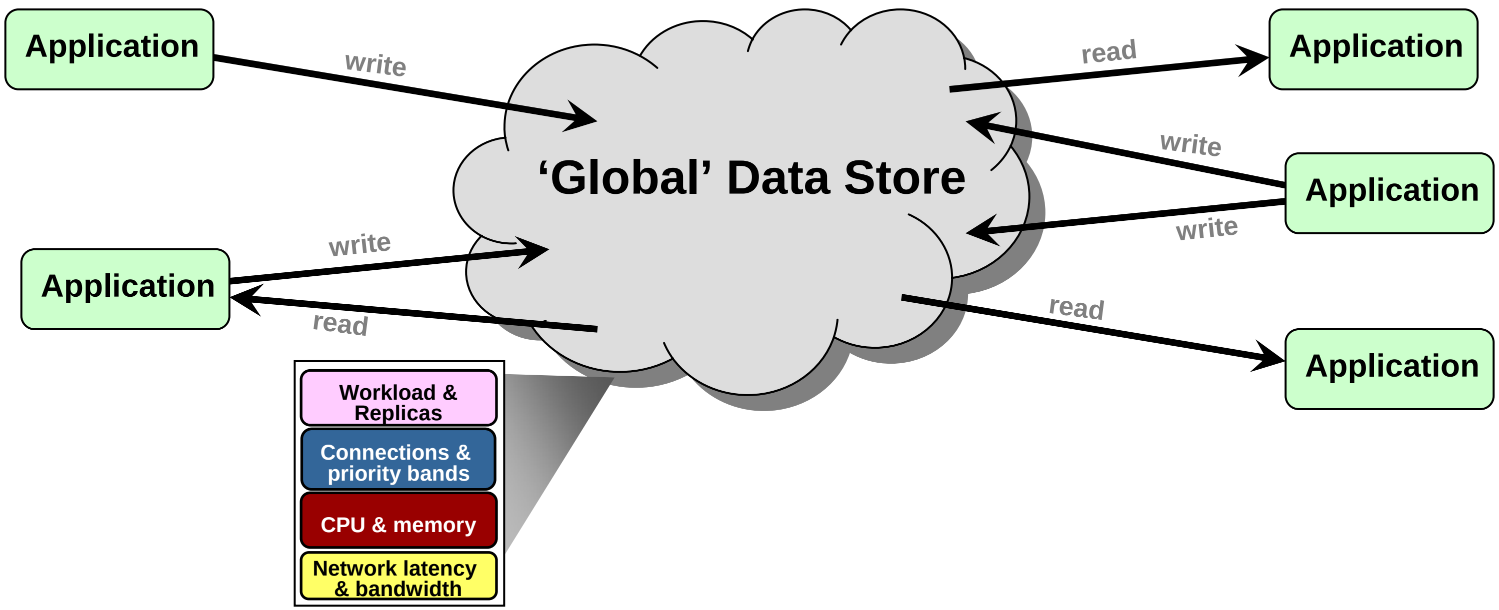




"Harder" Requirements

"Softer" Requirements

Promising Approach: The OMG Data Distribution Service (DDS)

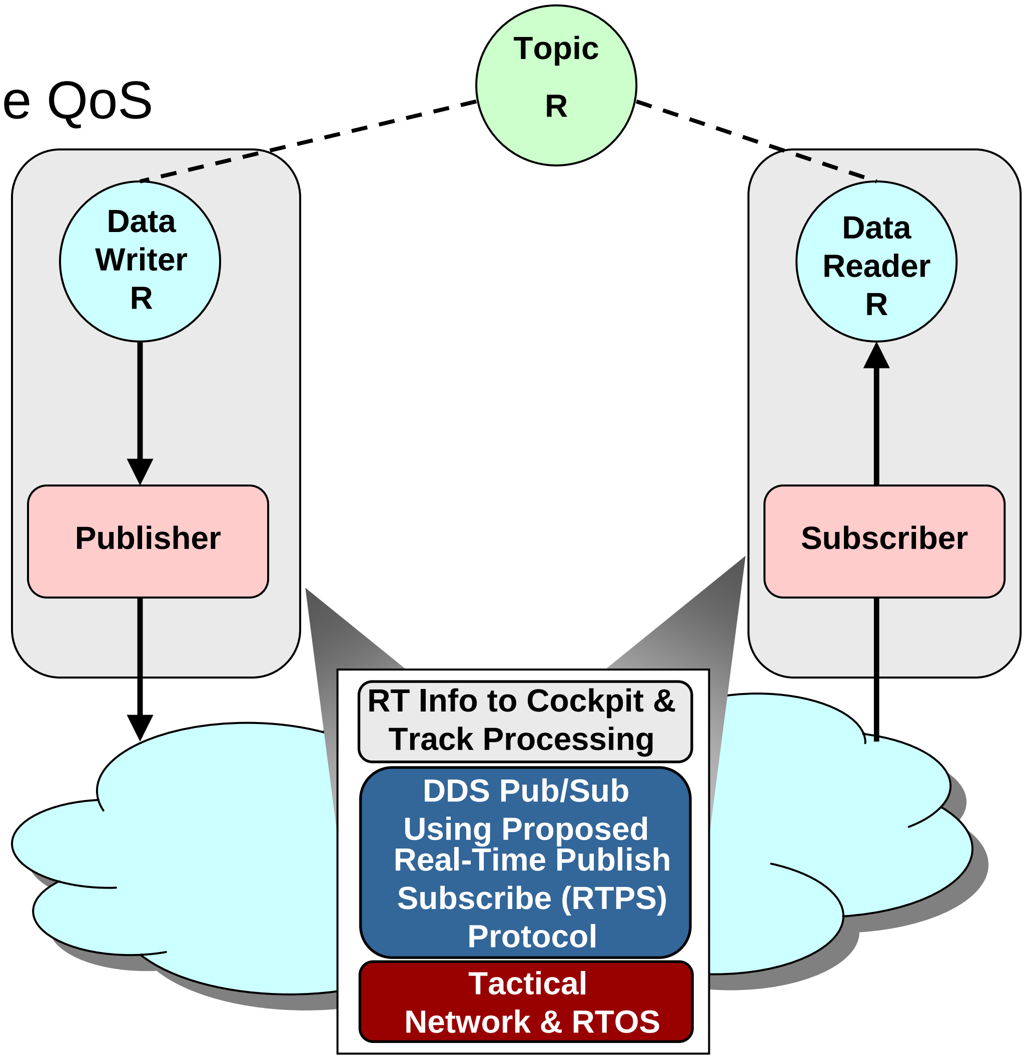


Provides flexibility, power & modular structure by decoupling:

- **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**

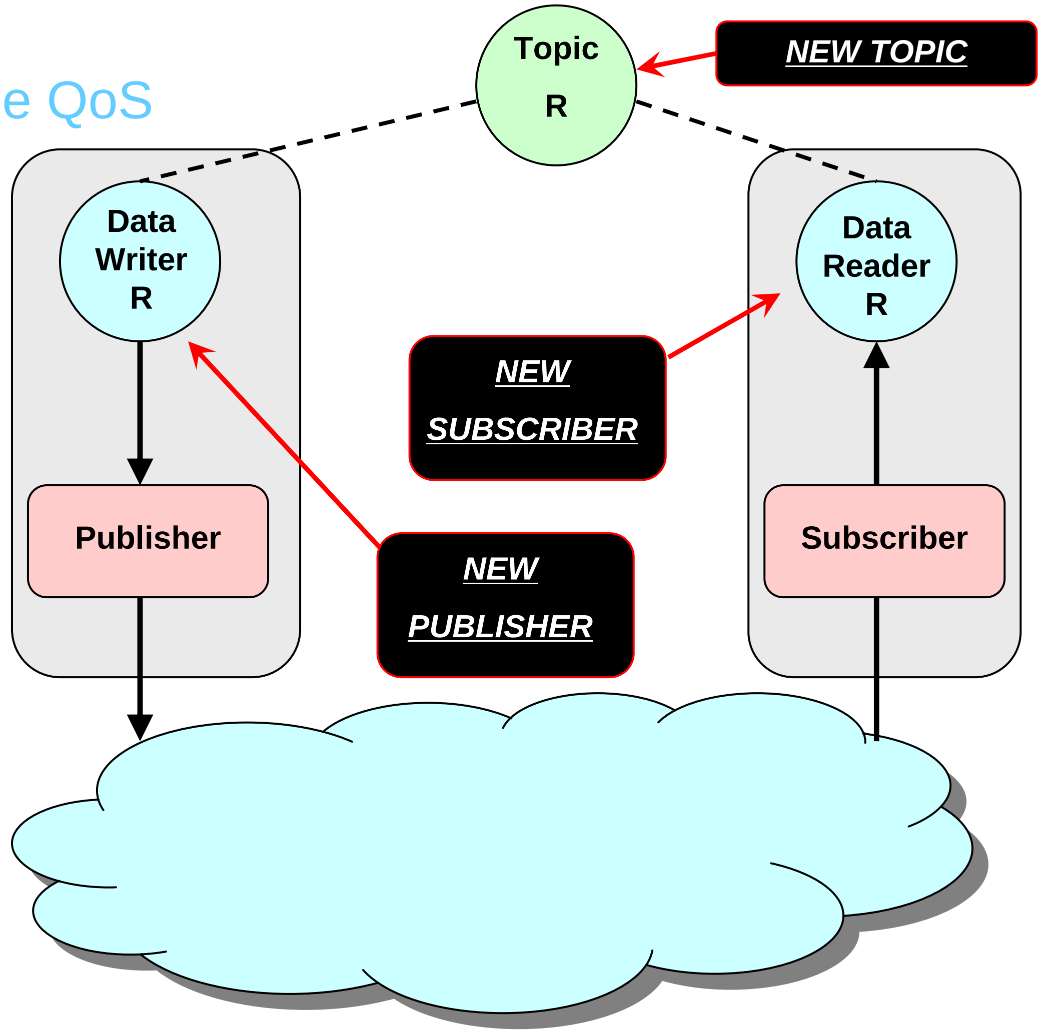
Overview of the Data Distribution Service (DDS)

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



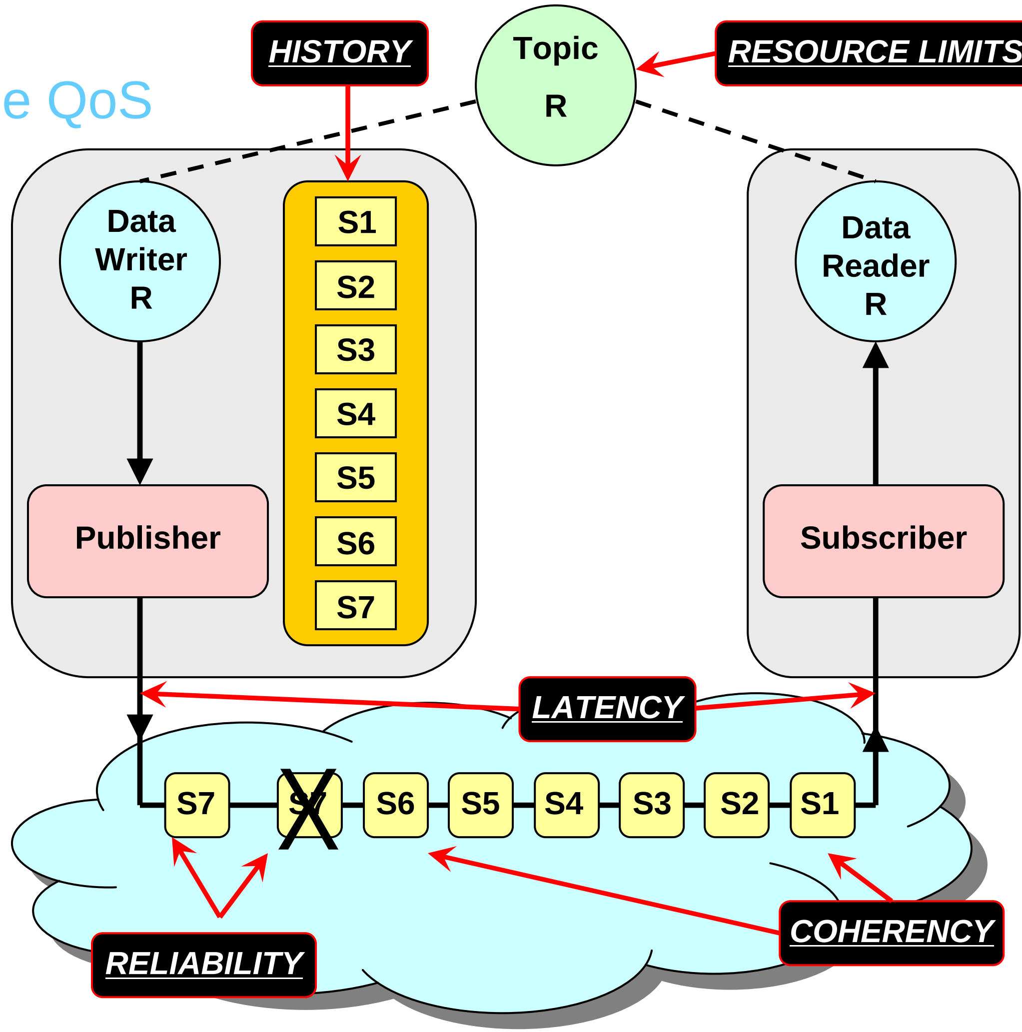
Overview of the Data Distribution Service (DDS)

- 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



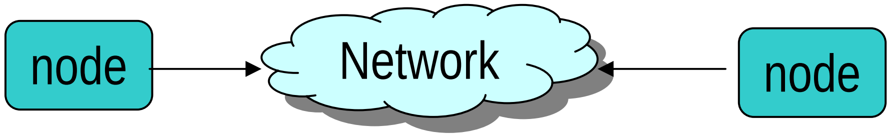
Overview of the Data Distribution Service (DDS)

- 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



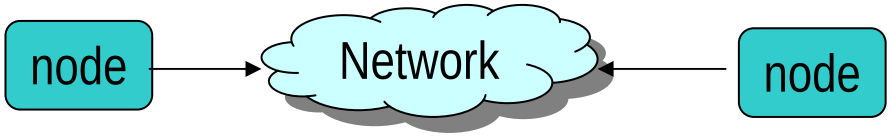
Overview of DDS Implementation Architectures

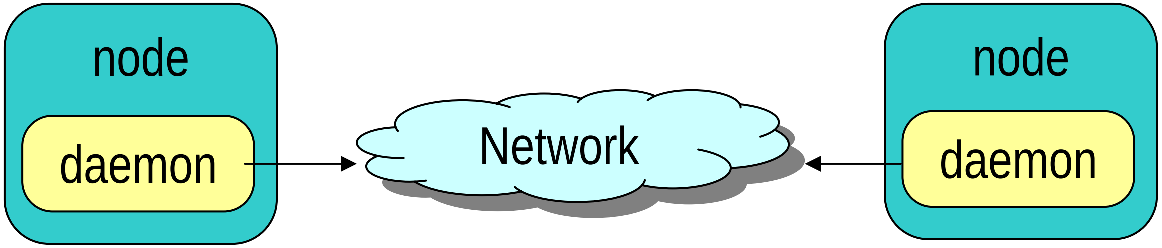
- **Decentralized Architecture**
  - embedded threads to handle communication, reliability, QoS etc



Overview of DDS Implementation Architectures

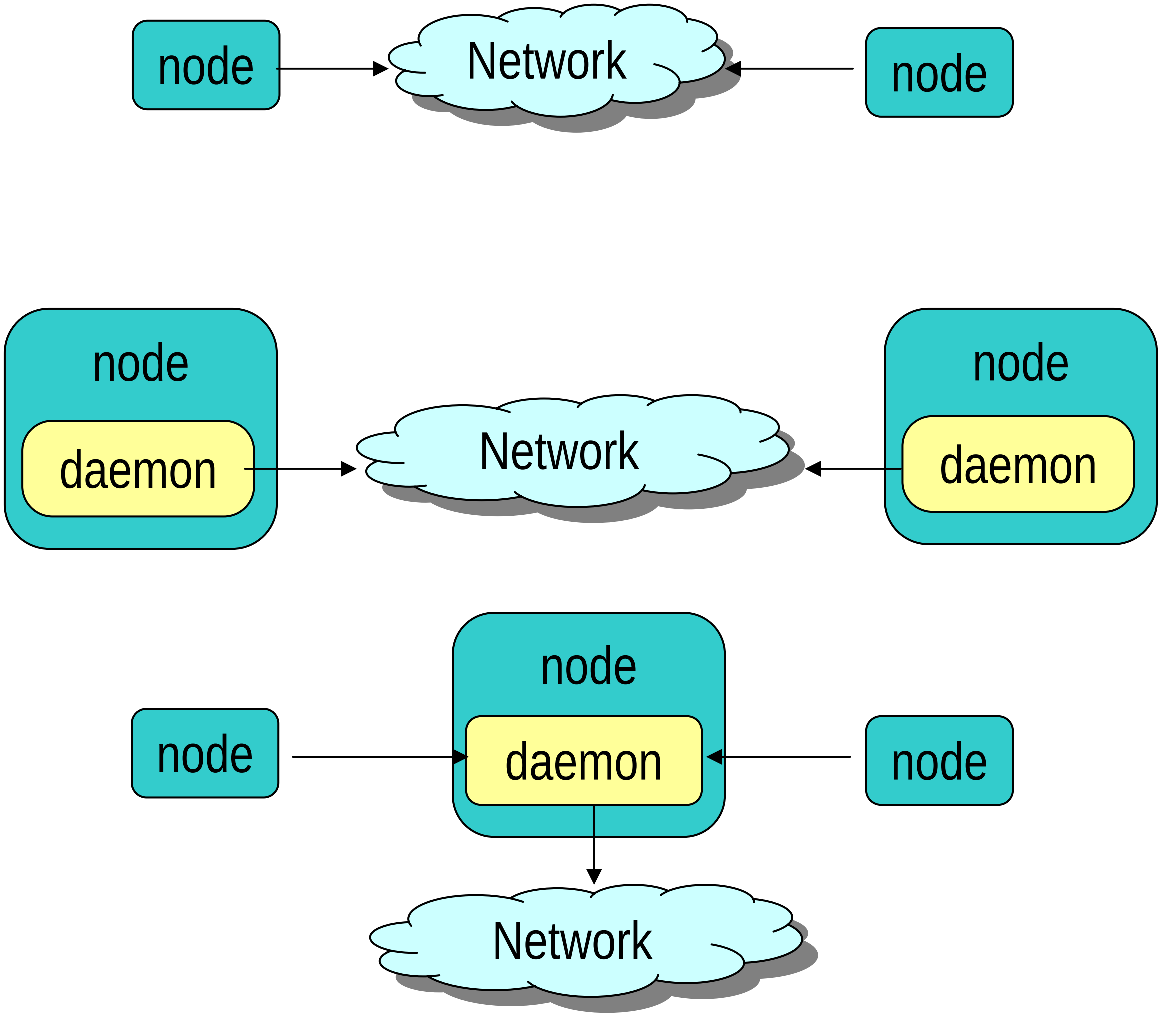
- **Decentralized Architecture**
  - embedded threads to handle communication, reliability, QoS etc
- **Federated Architecture**
  - a separate daemon process to handle communication, reliability, QoS, etc.



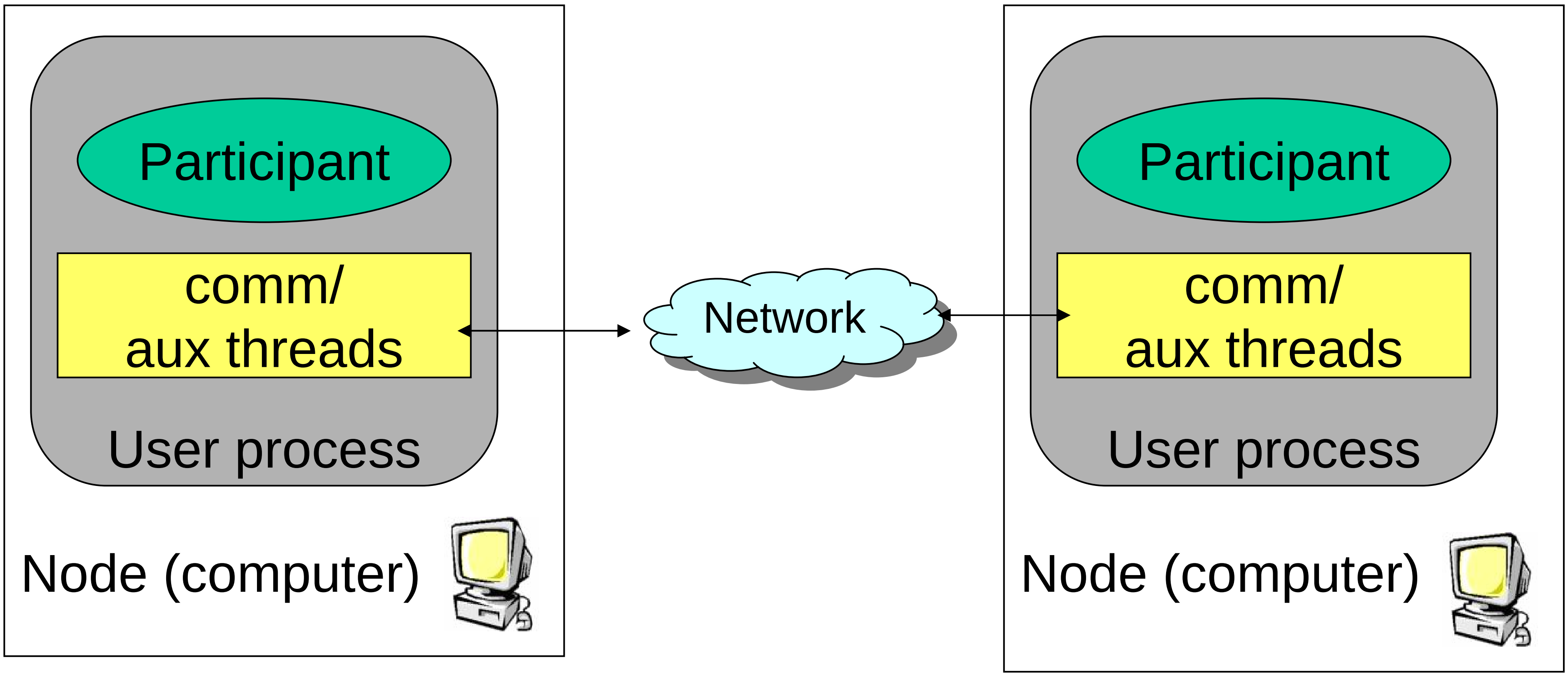


Overview of DDS Implementation Architectures

- **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



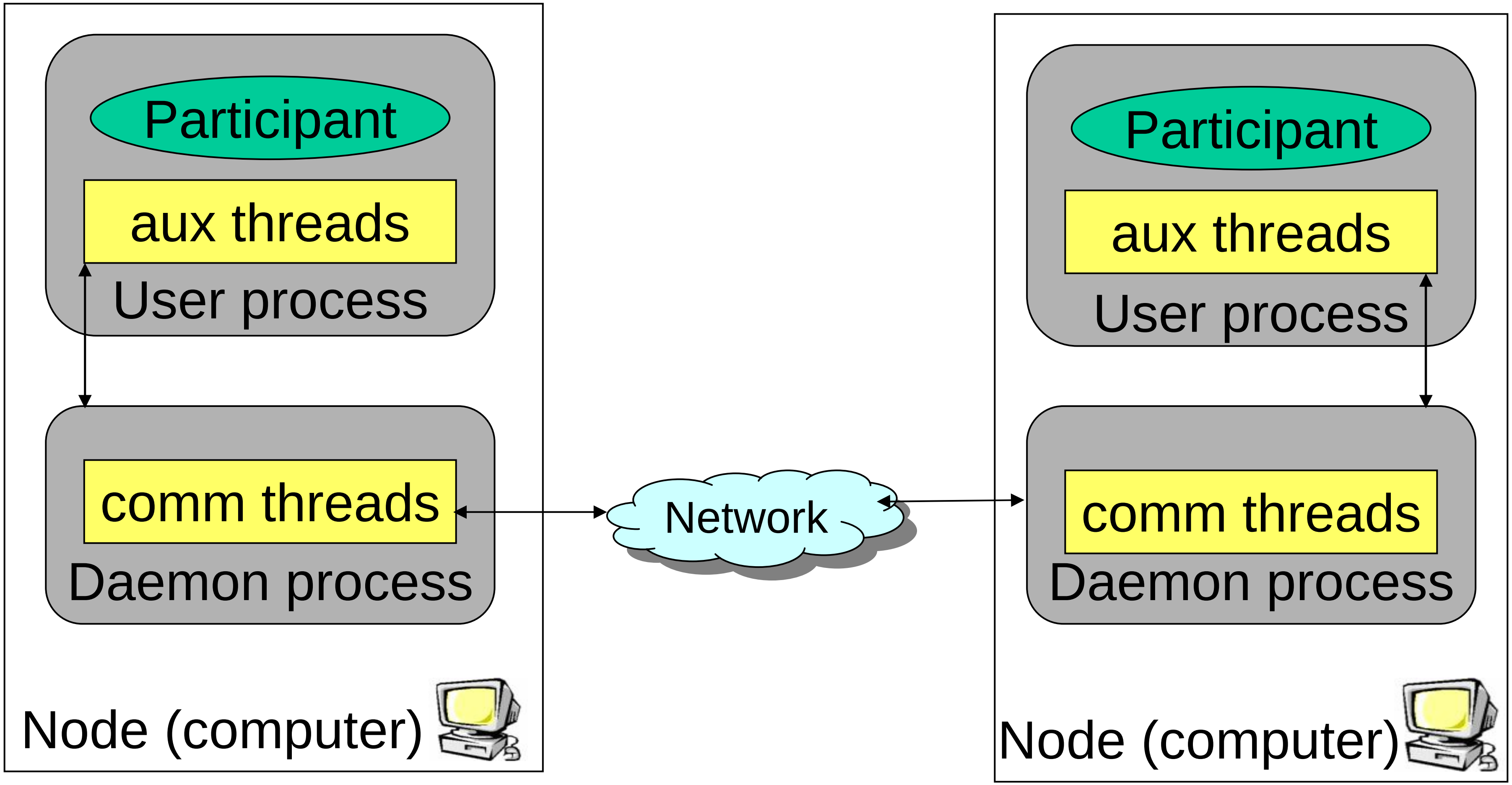
DDS1 (Decentralized Architecture)



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

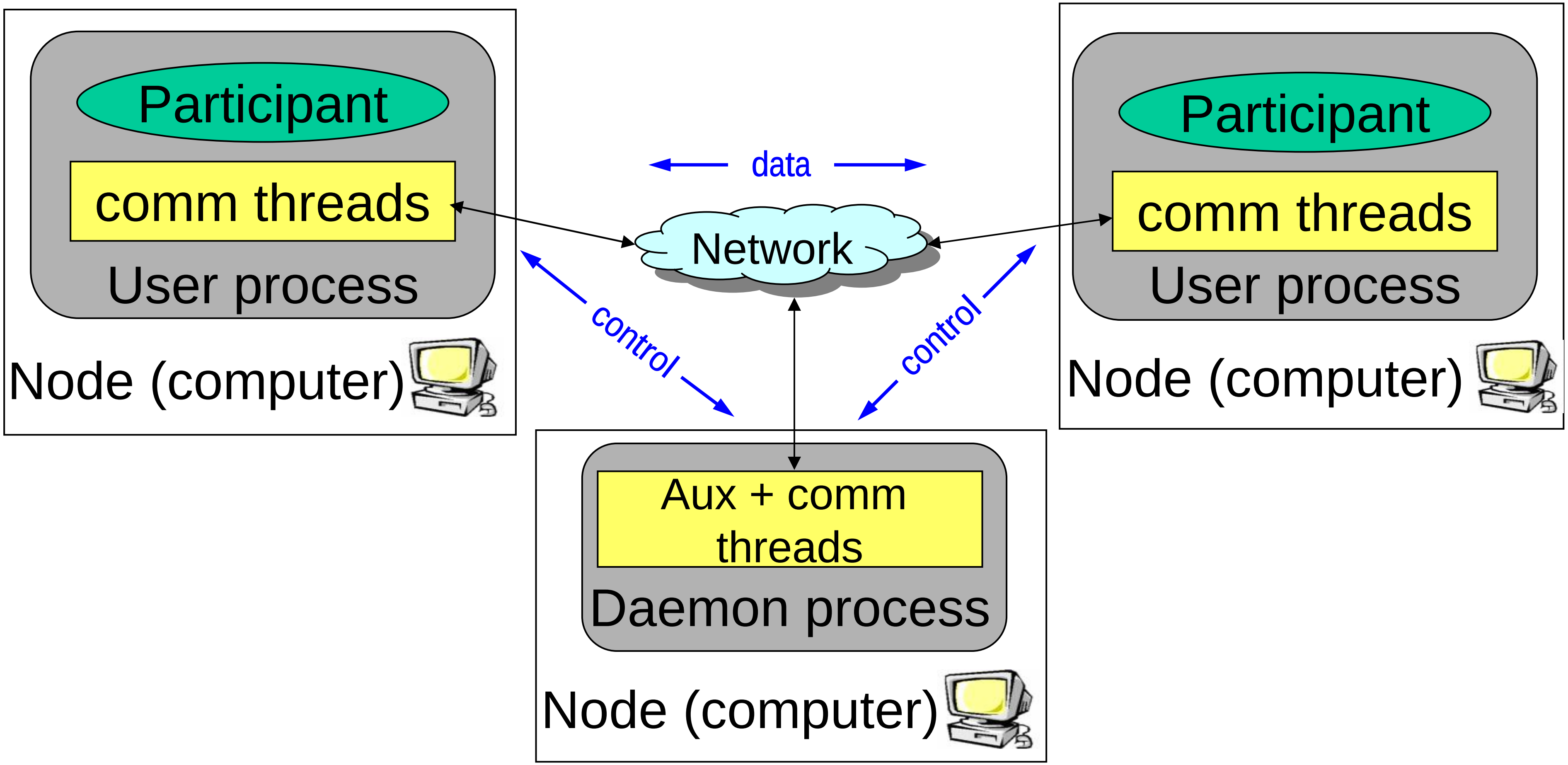
Cons: User process more complex, e.g., must handle config details (efficient discovery, multicast)

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)

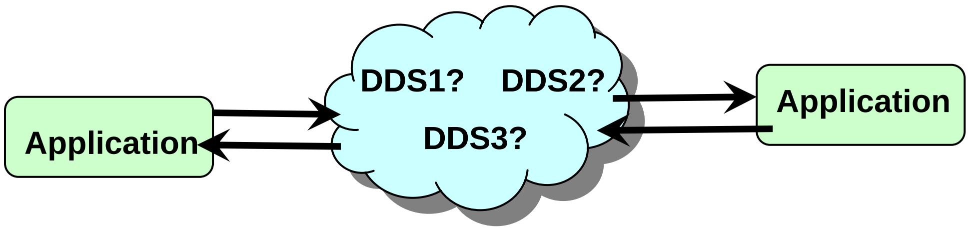
QoS	Description	DDS1	DDS2	DDS3
DURABILITY	Controls how long published samples are stored by the middleware for late-joining data readers	VOLATILE TRANSIENT-LOCAL	VOLATILE TRANSIENT-LOCAL TRANSIENT PERSISTENT	VOLATILE
HISTORY	Sets number of samples that DDS will store locally for data writers & data readers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

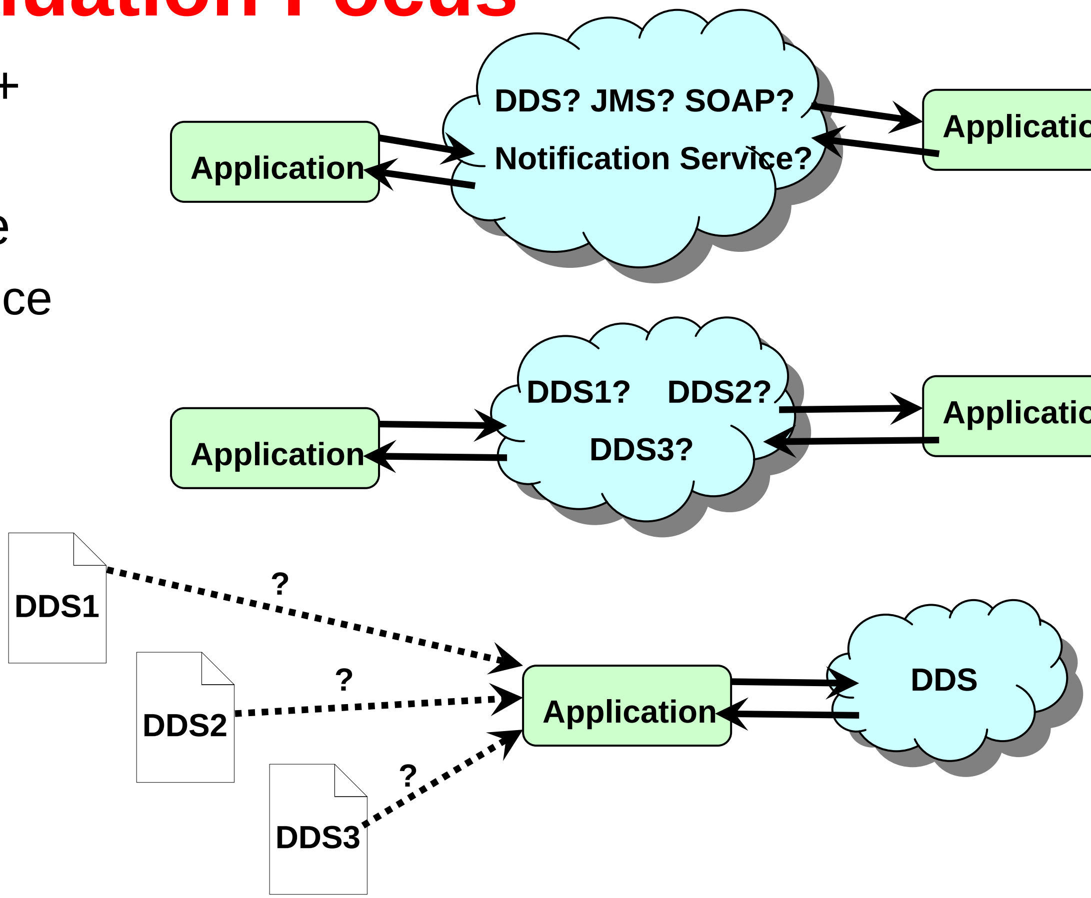
Evaluation Focus

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



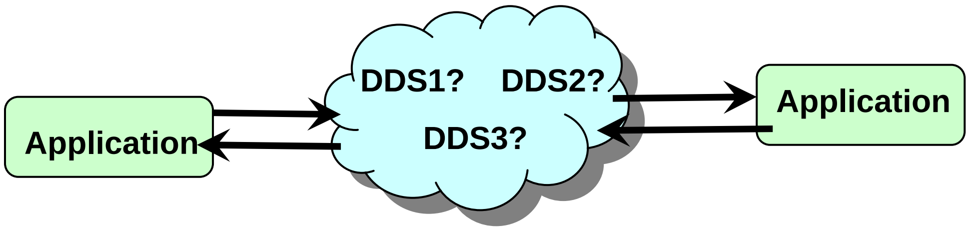
Evaluation Focus

- 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

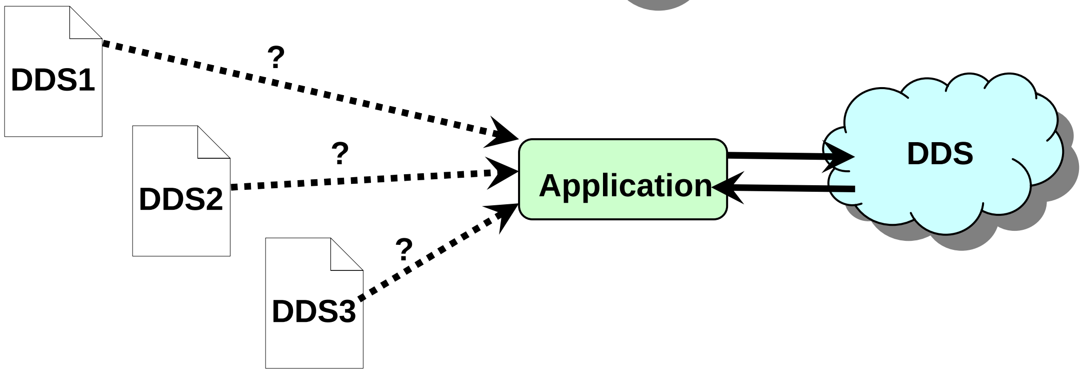


Evaluation Focus

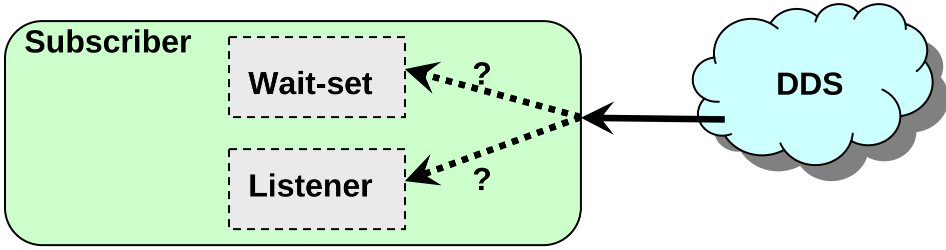
- Compare performance of C++ implementations of DDS to:
  - Other pub/sub middleware
    - CORBA Notification Service
    - SOAP
    - Java Messaging Service
  - Each other



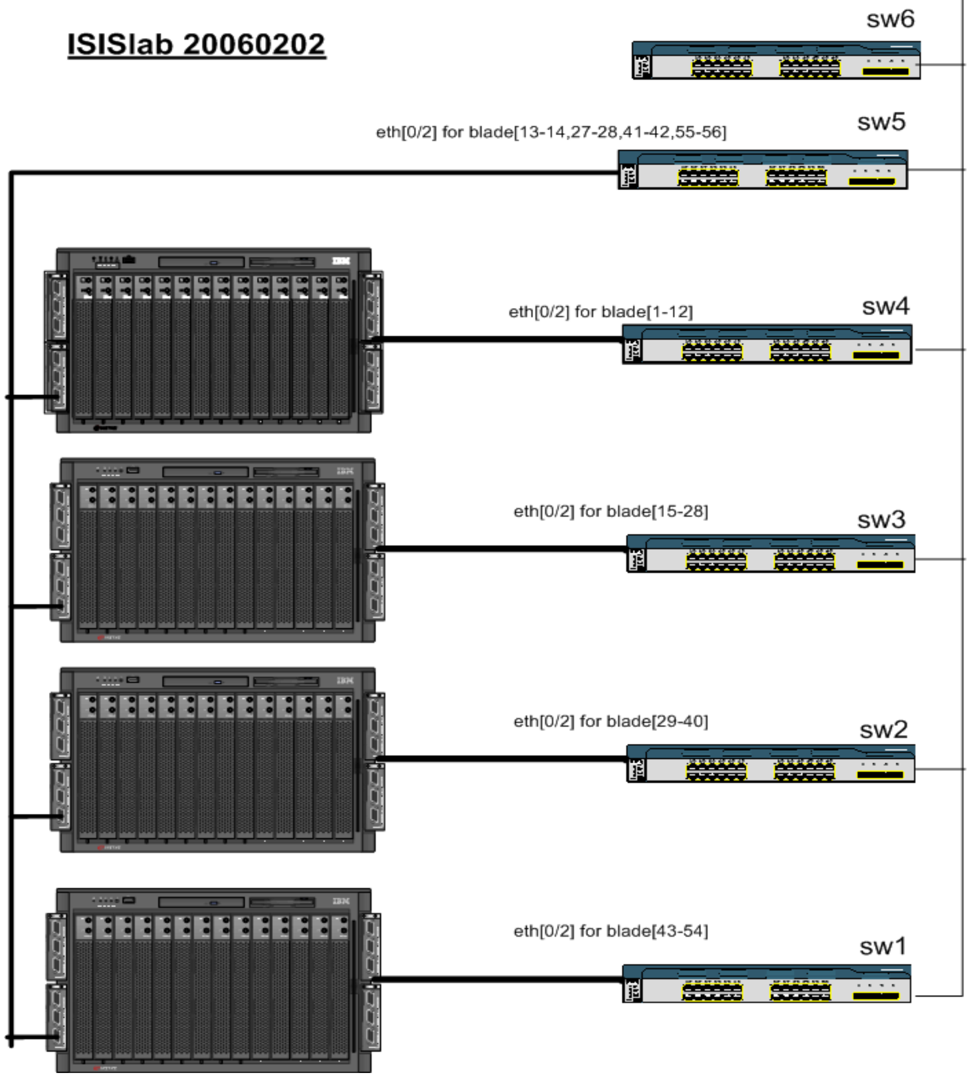
- Compare DDS portability & configuration details



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



Overview of ISISlab Testbed



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

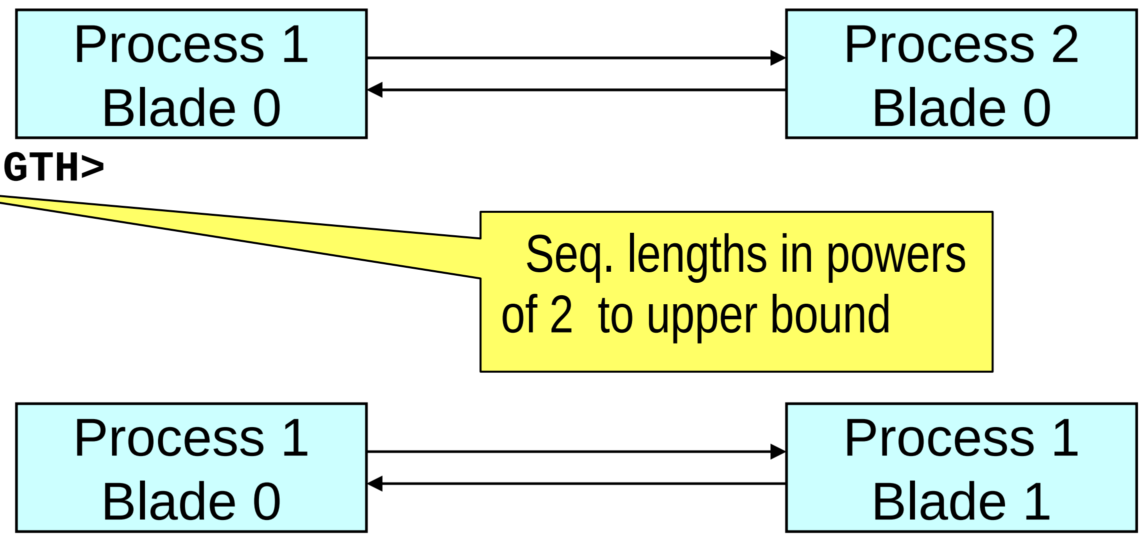
Benchmarking Challenges

- 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 Latency And Jitter

```
const short MAX_MSG_LENGTH =
  16384;
struct PubMessage {
  long seqnum;
  sequence<octet,MAX_MSG_LENGTH>
    data;
};
struct AckMessage {
  long seqnum;
};
```

Latency & jitter on same node



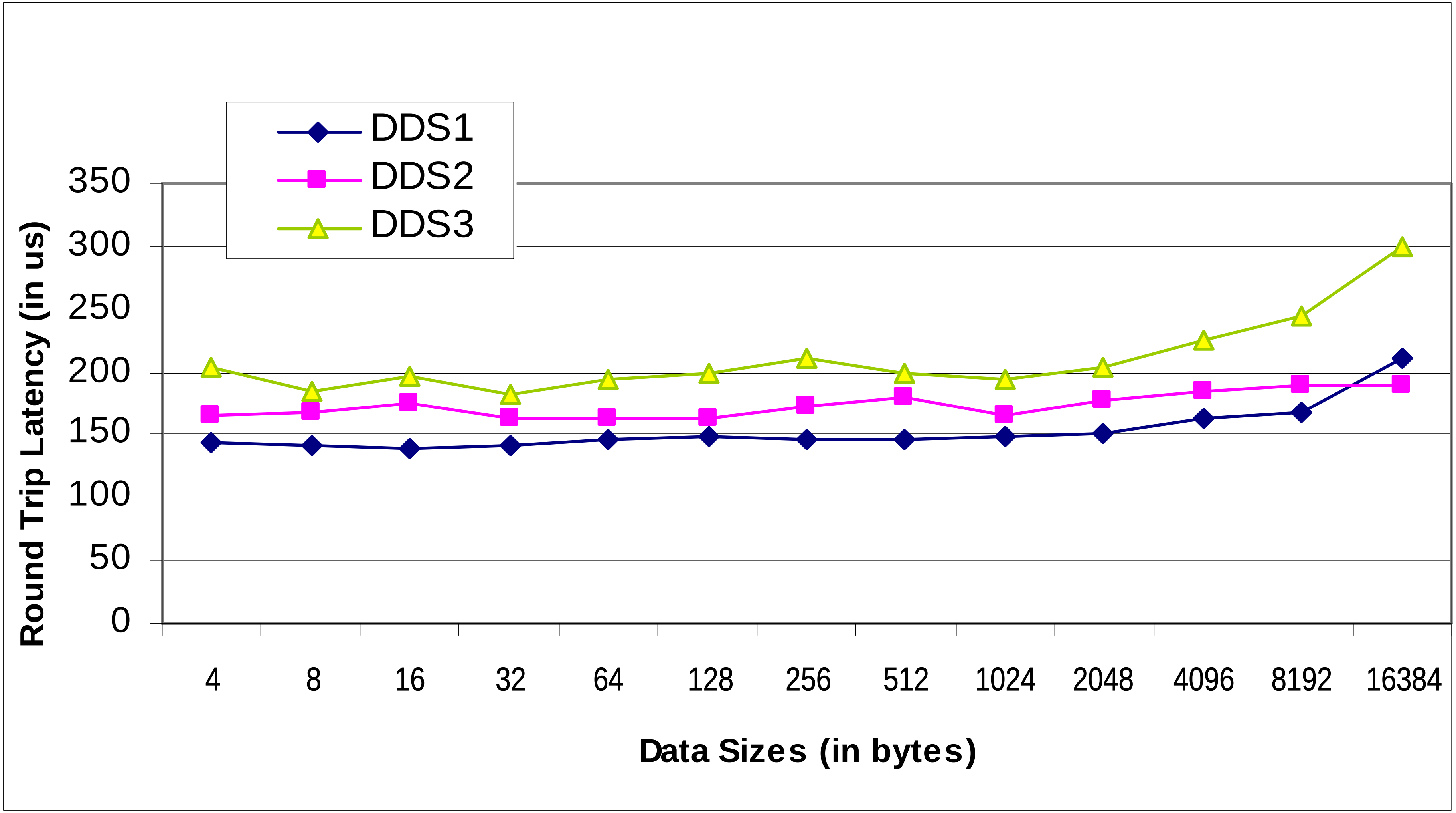
Seq. lengths in powers of 2 to upper bound

Ack message of 4 bytes

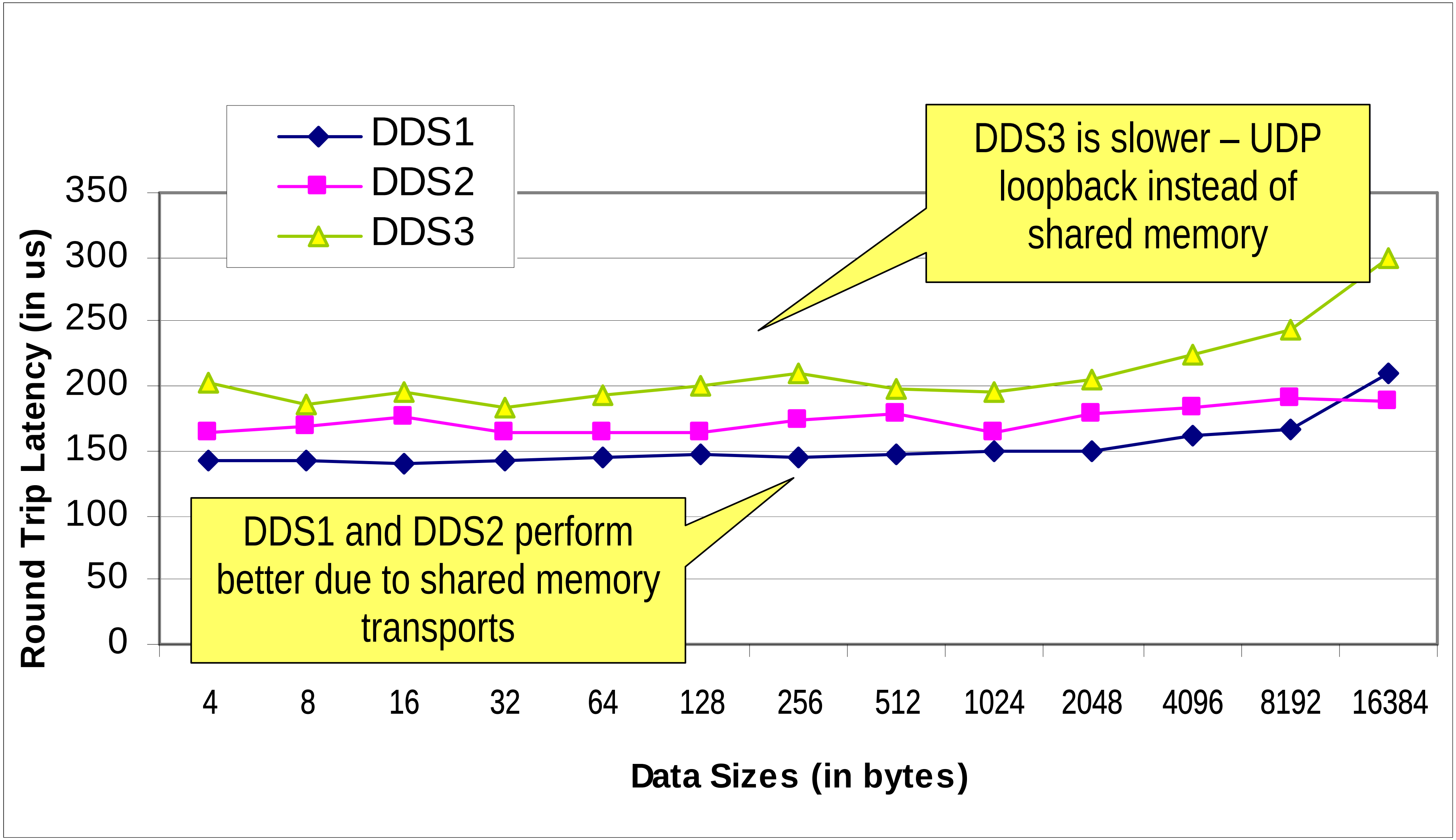
Tested seq. of bytes

Latency & jitter on different nodes

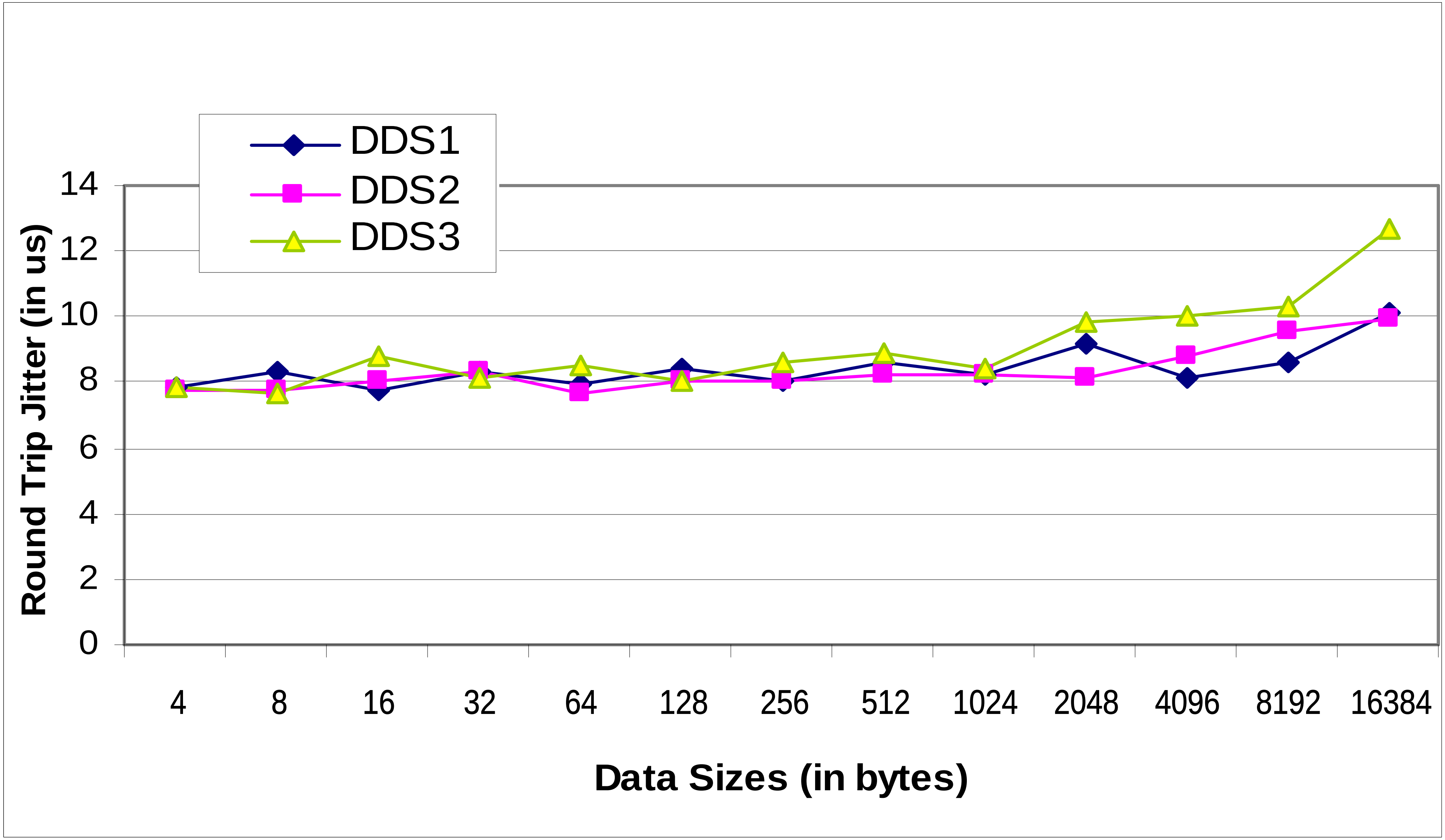
1-to-1 Single Node Latency



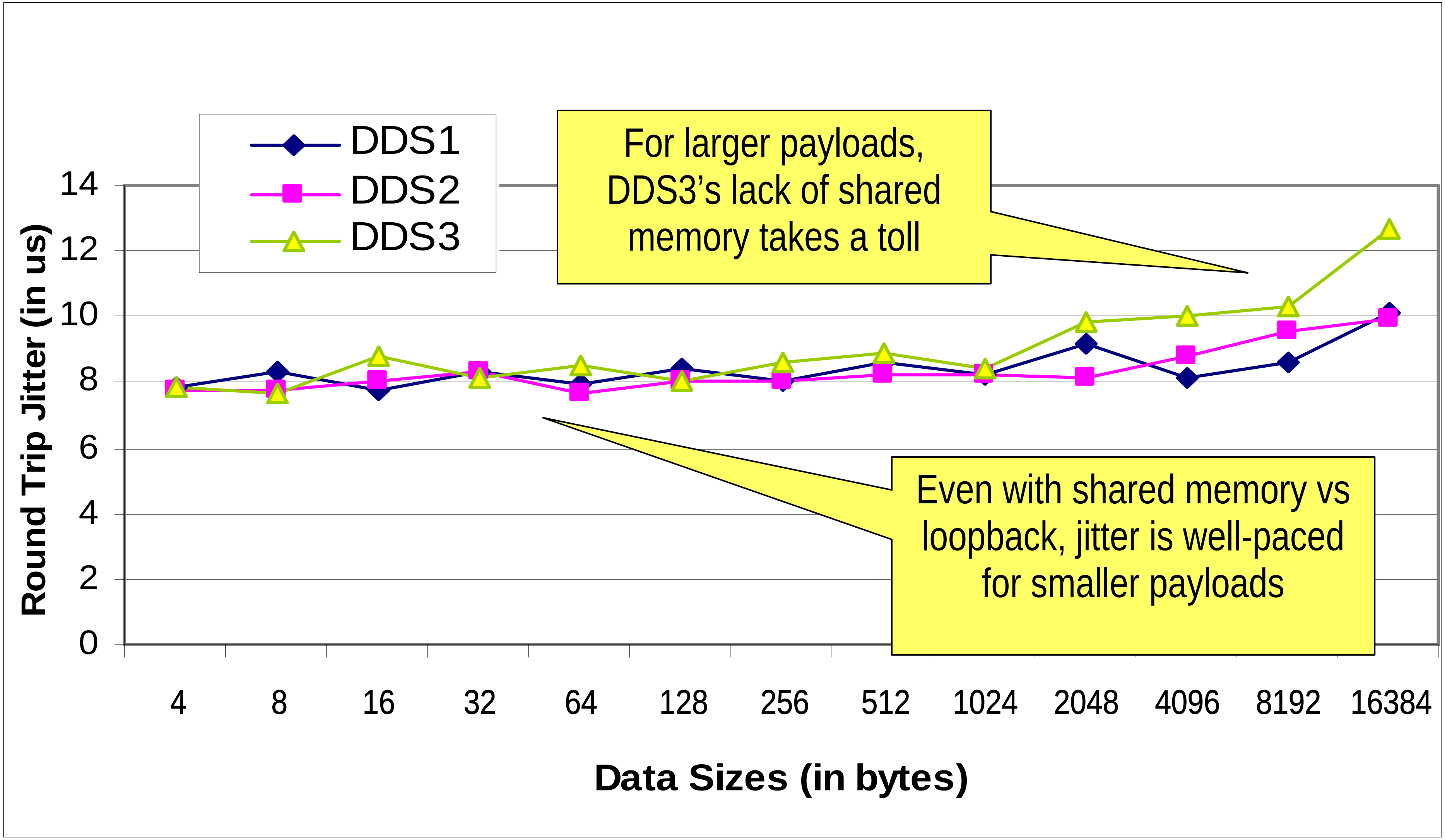
1-to-1 Single Node Latency



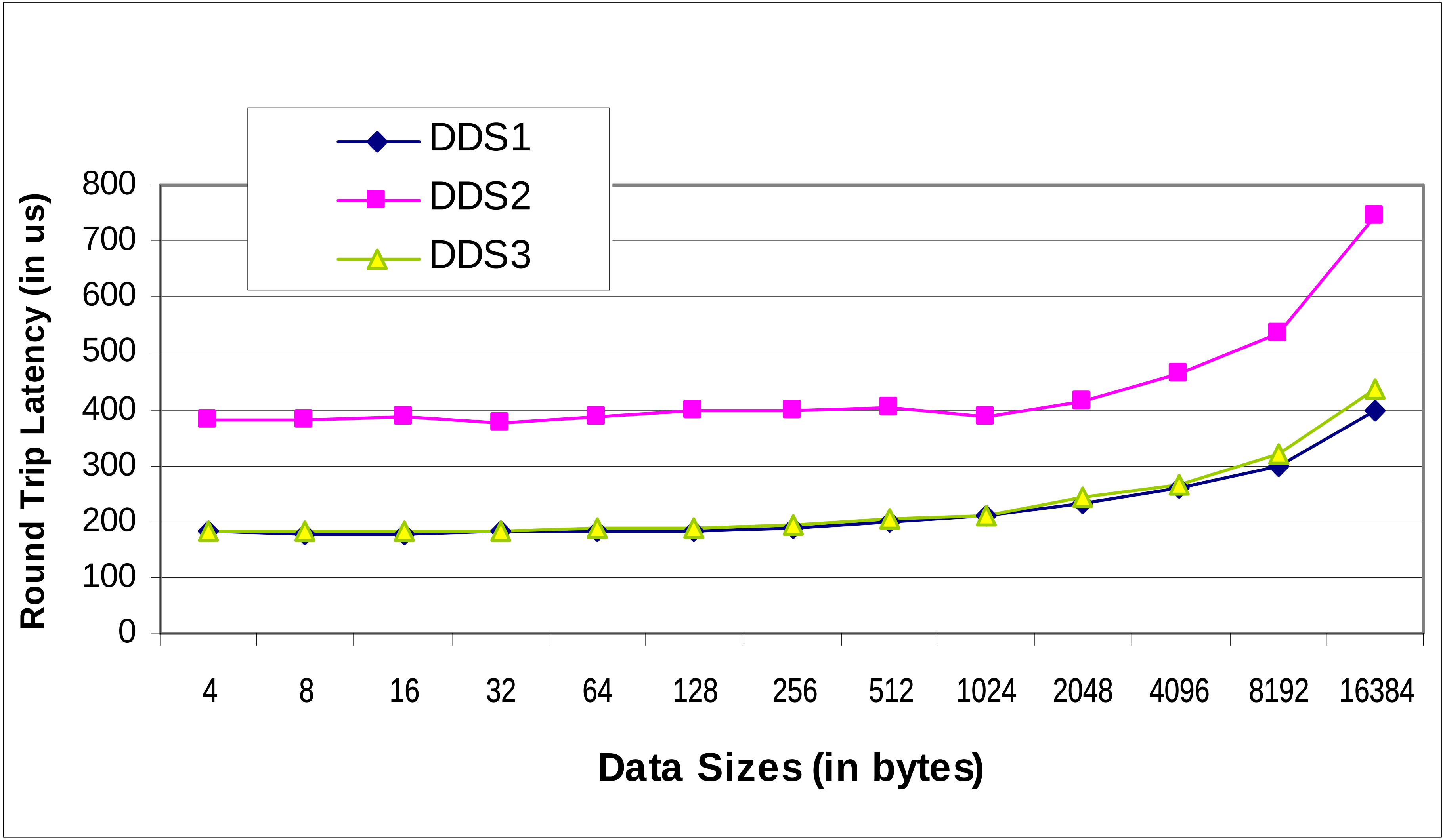
1-to-1 Single Node Jitter



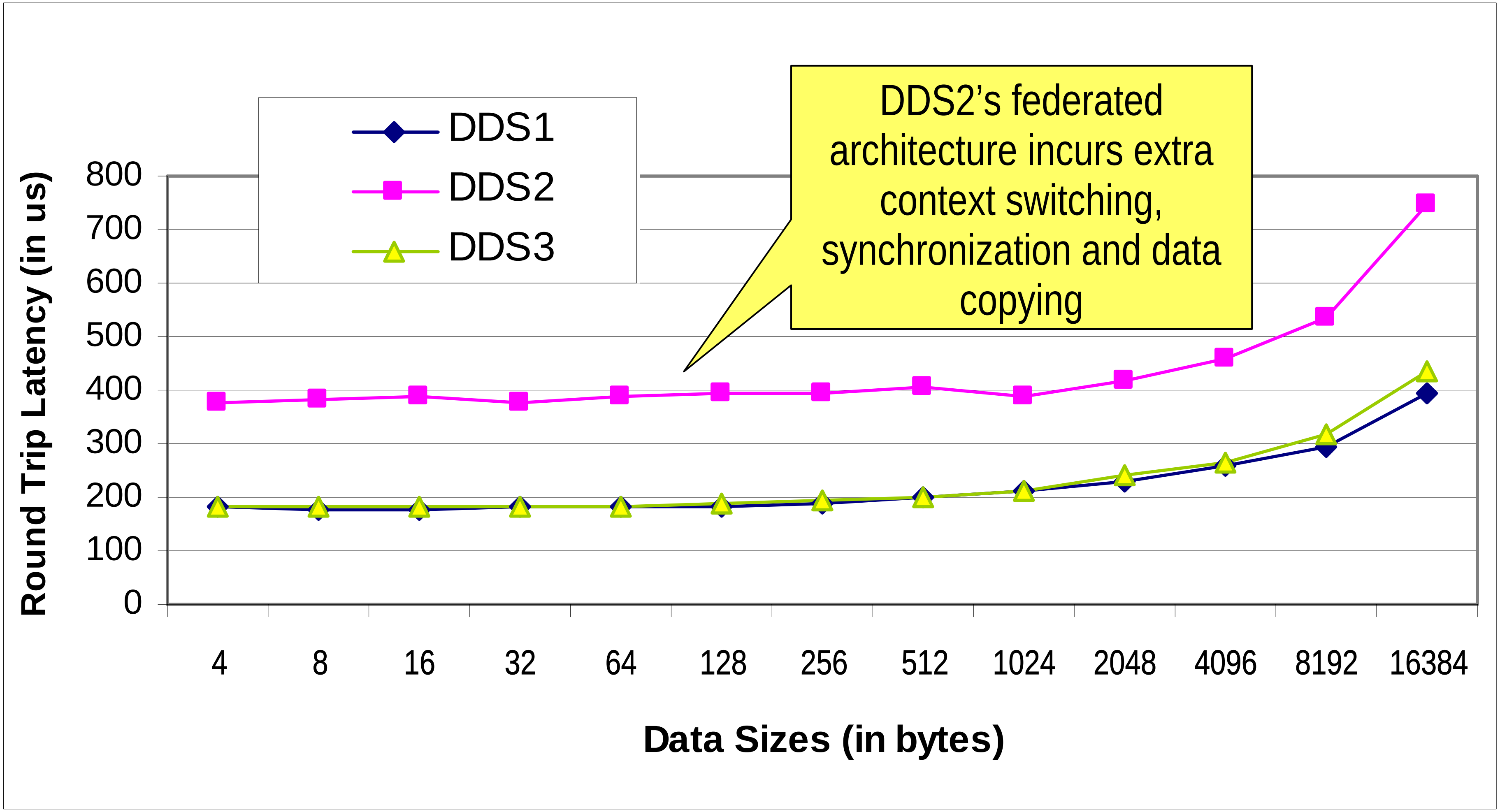
1-to-1 Single Node Jitter



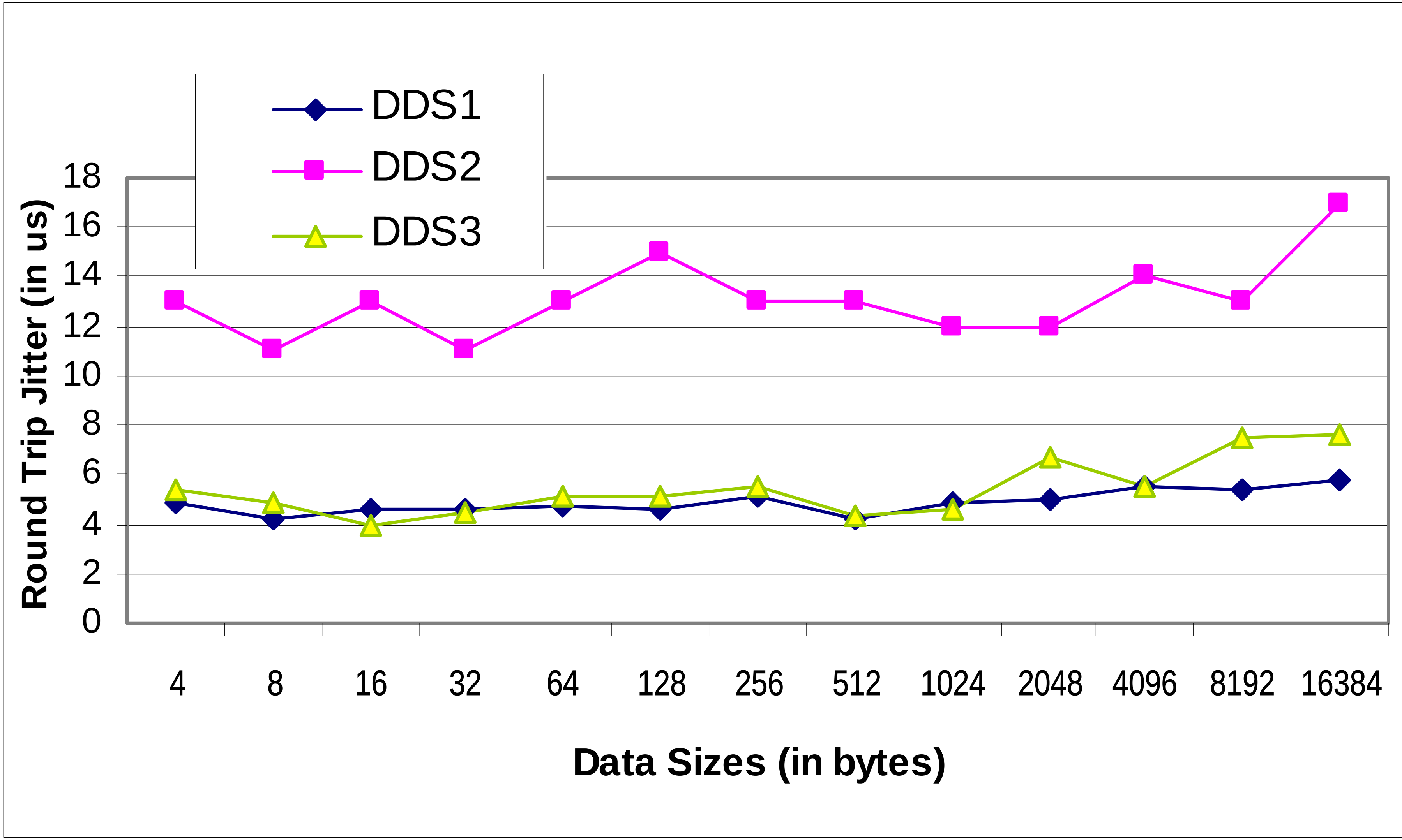
1-to-1 Multiple Node Latency



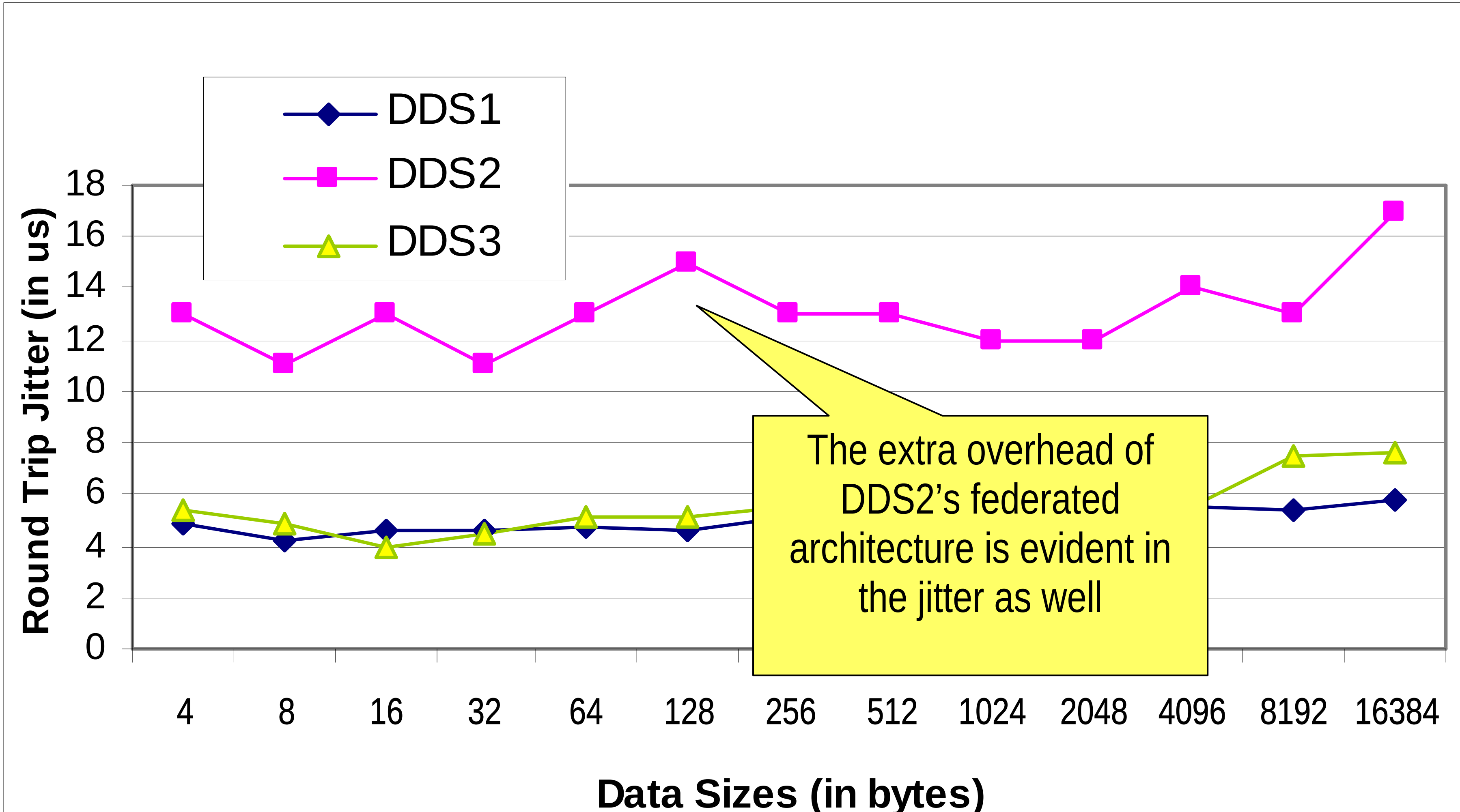
1-to-1 Multiple Node Latency



1-to-1 Multiple Node Jitter



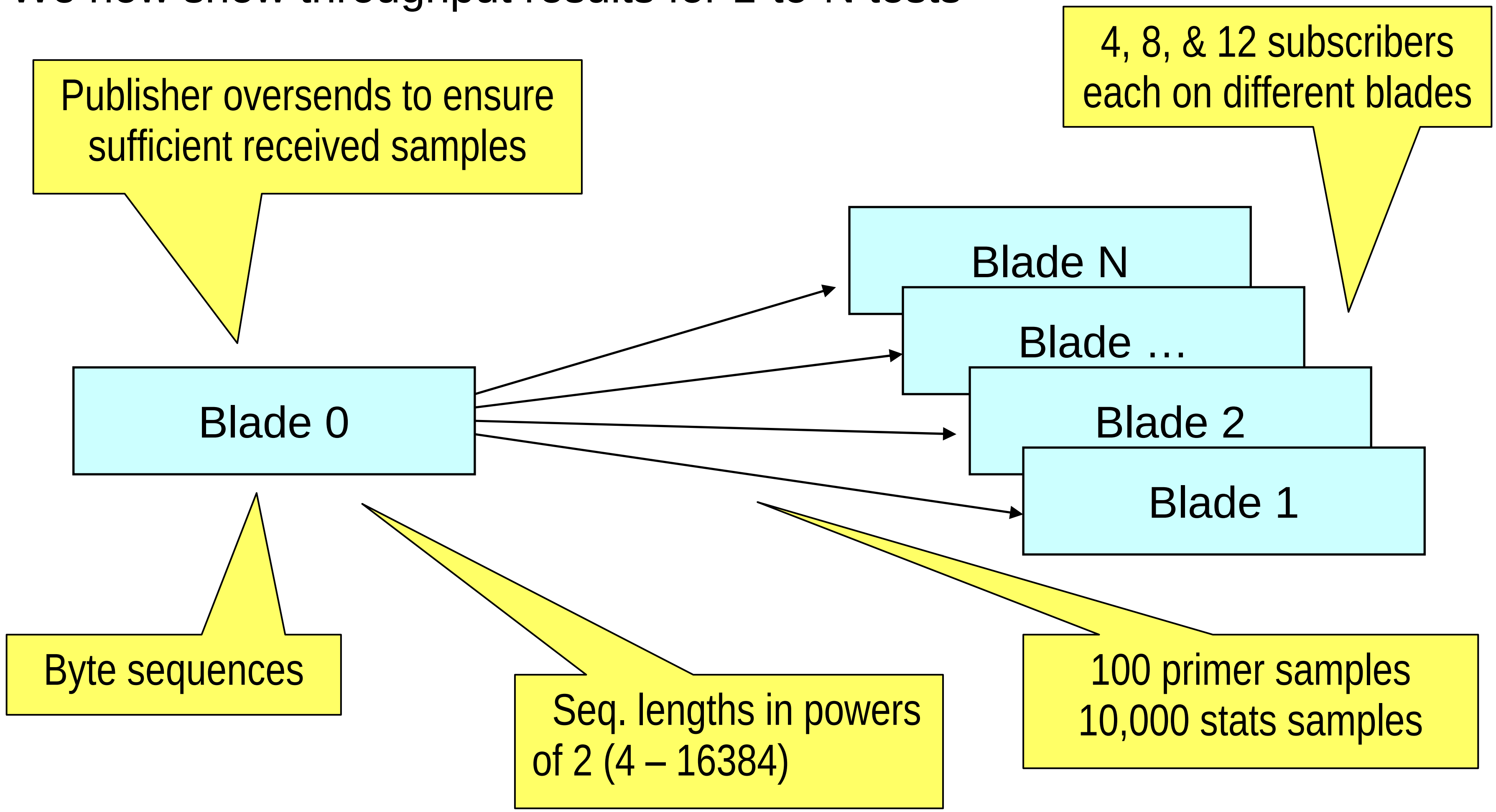
1-to-1 Multiple Node Jitter



Federated arch. is designed for scalabilty, low CPU usage, not low latency

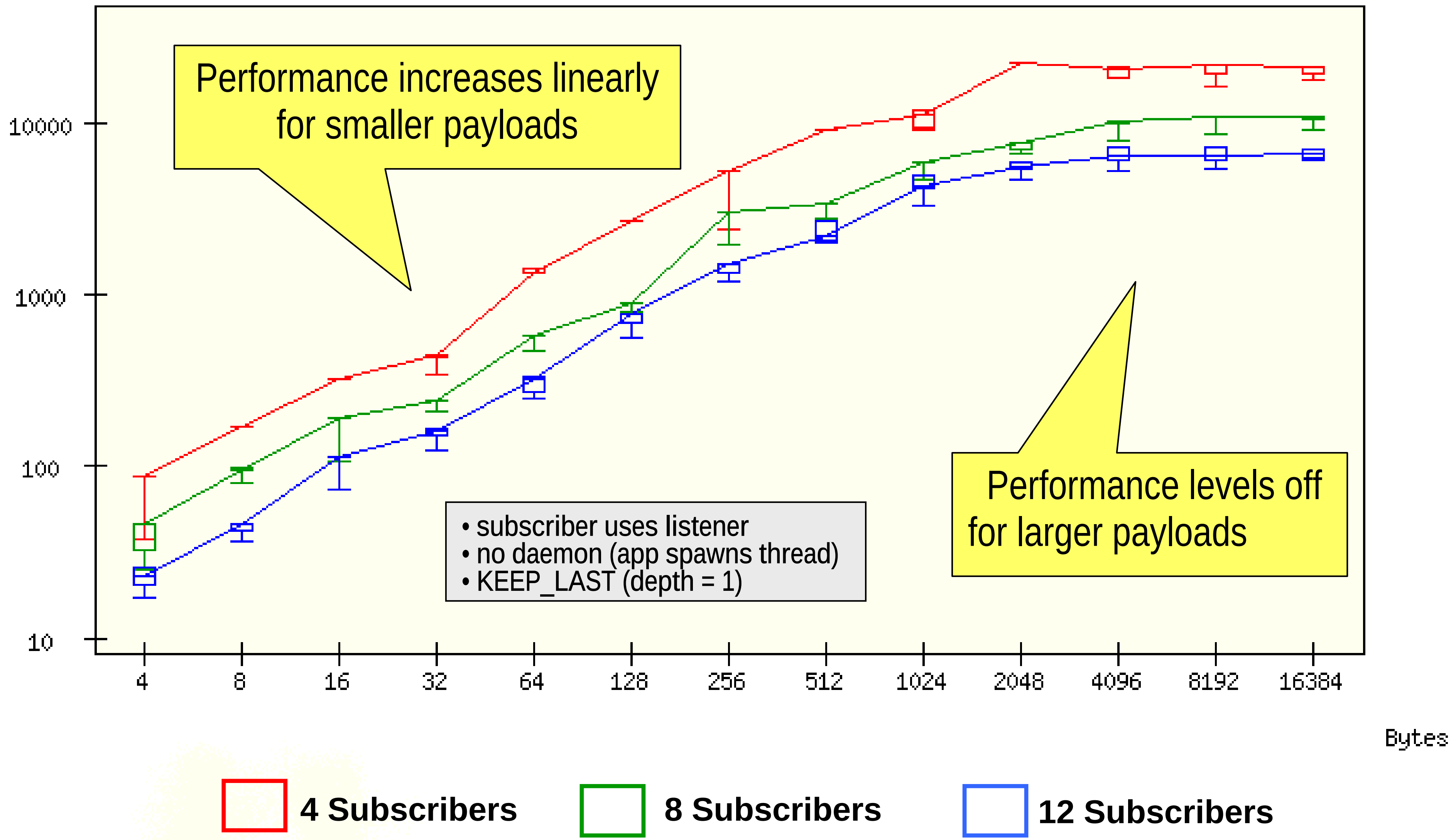
Scaling Up DDS Subscribers

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

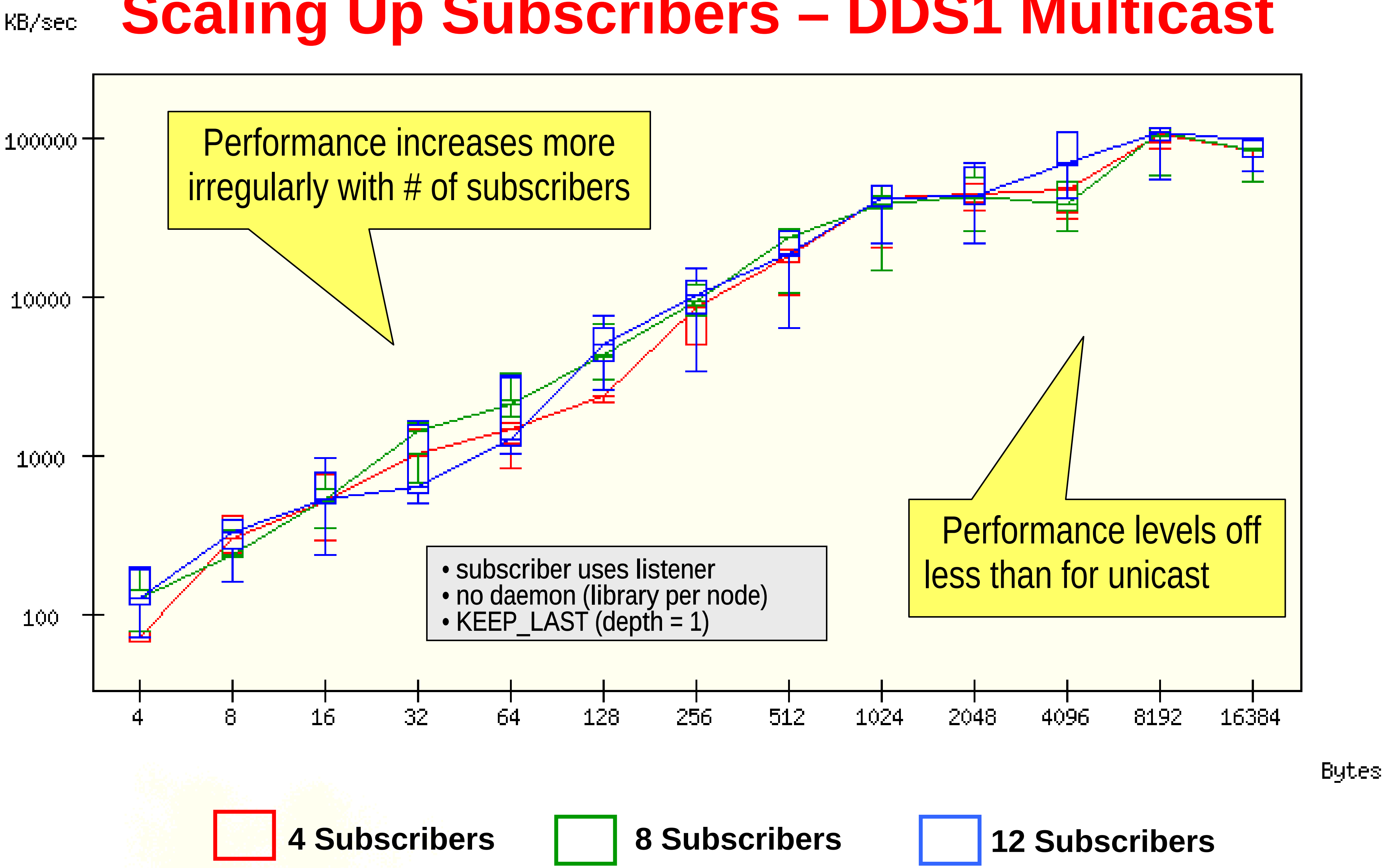


All following graphs plot median + “box-n-whiskers” (50%ile-min-max)

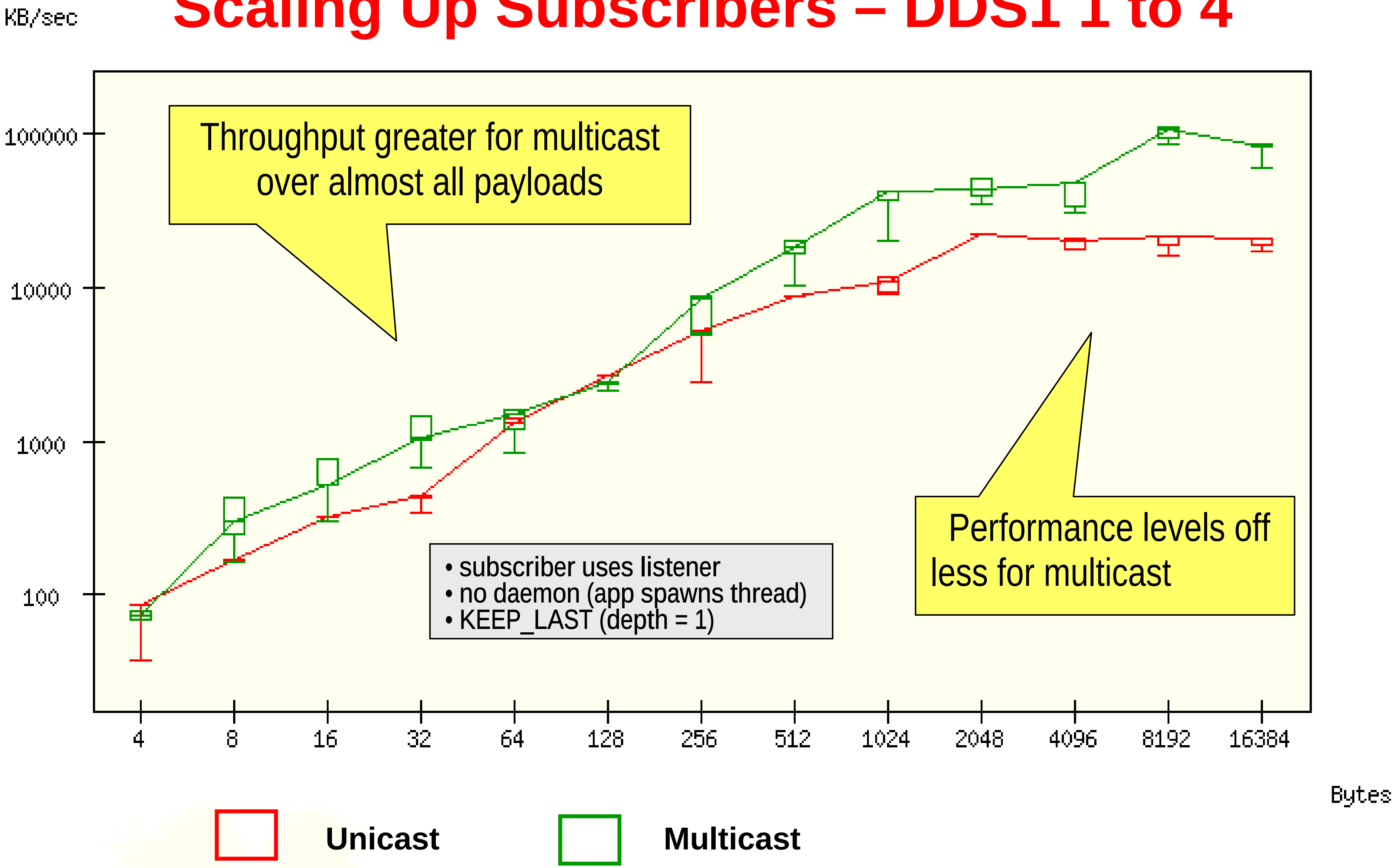
Scaling Up Subscribers – DDS1 Unicast



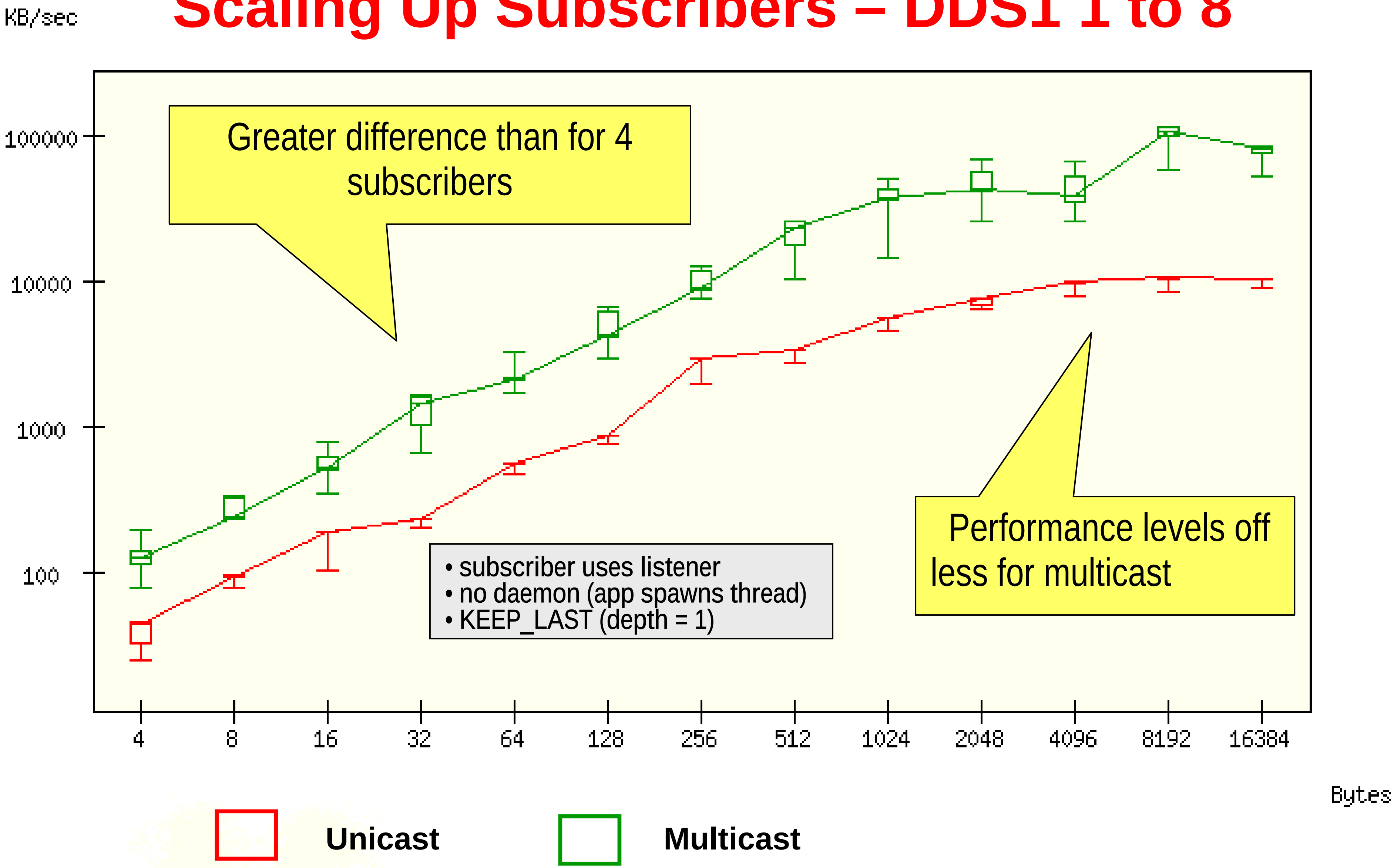
Scaling Up Subscribers – DDS1 Multicast



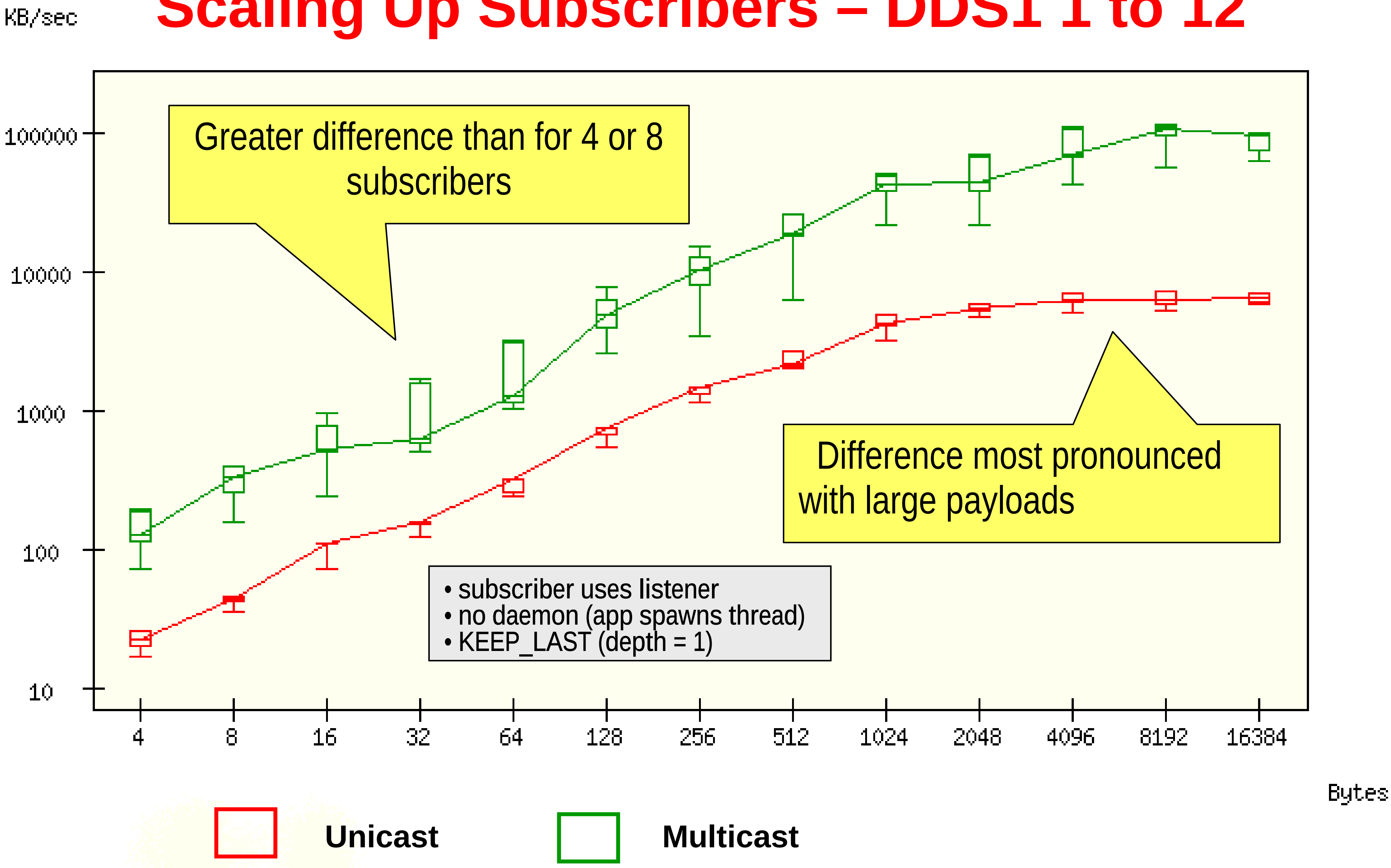
Scaling Up Subscribers – DDS1 1 to 4



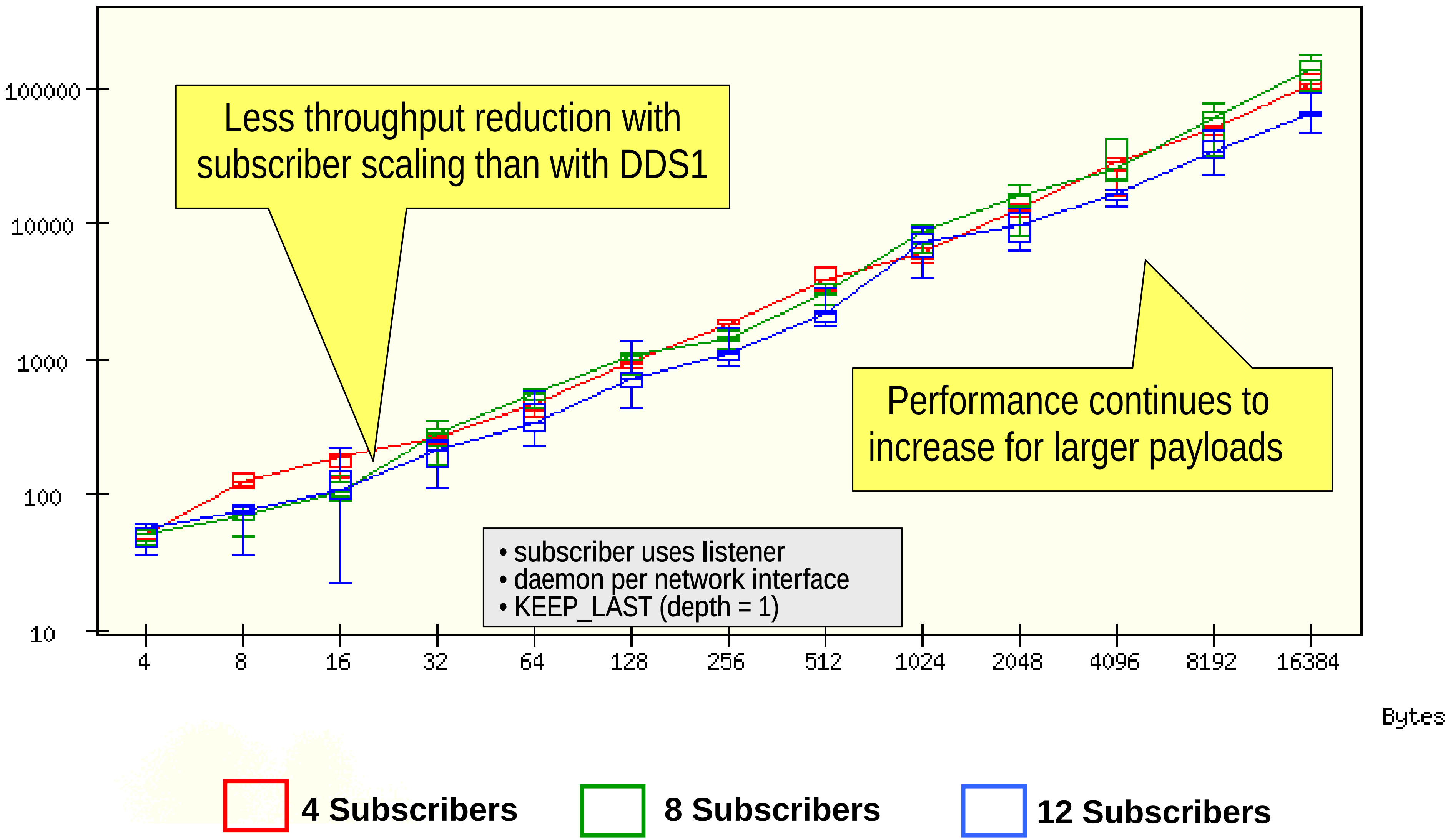
Scaling Up Subscribers – DDS1 1 to 8



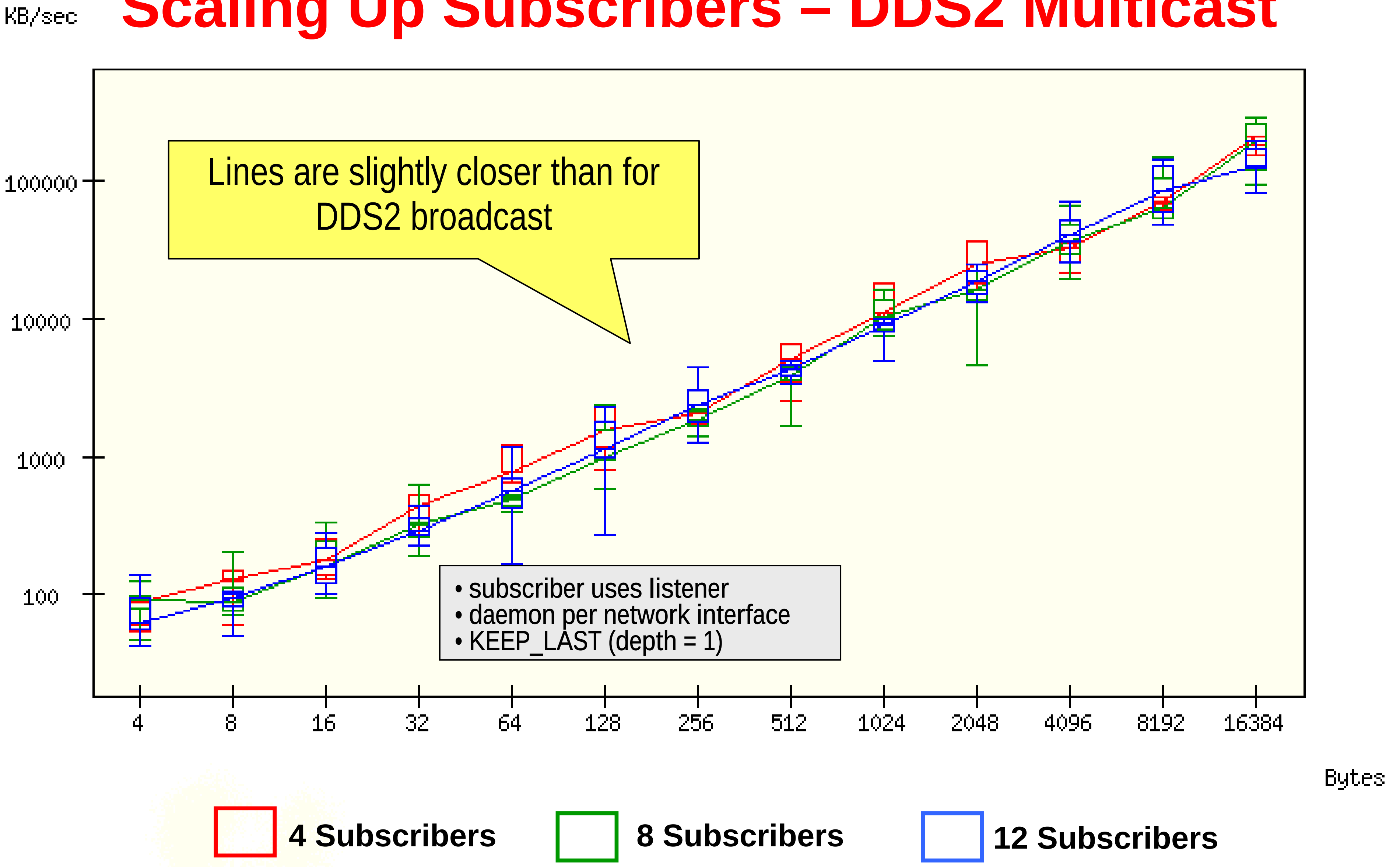
Scaling Up Subscribers – DDS1 1 to 12



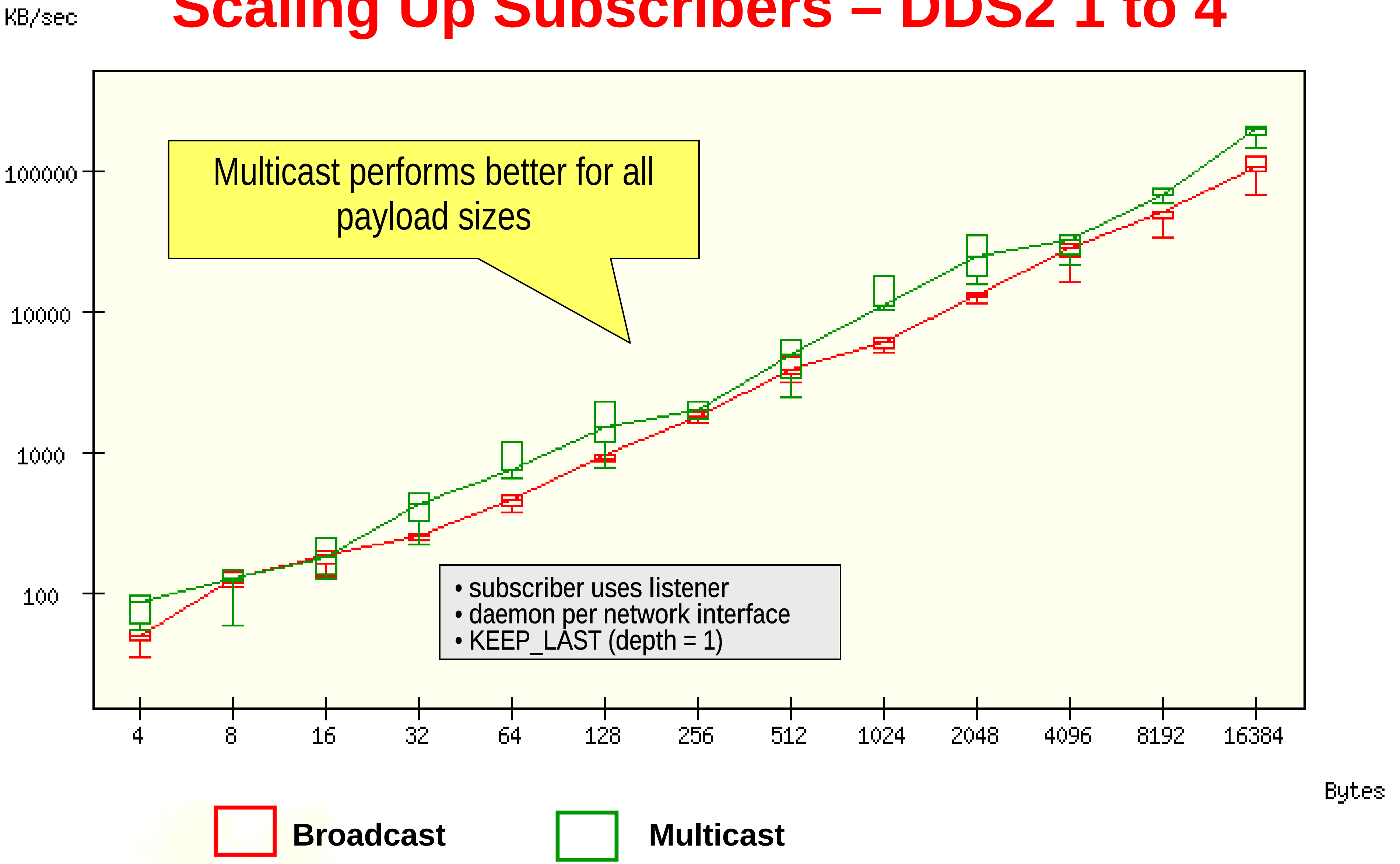
Scaling Up Subscribers – DDS2 Broadcast



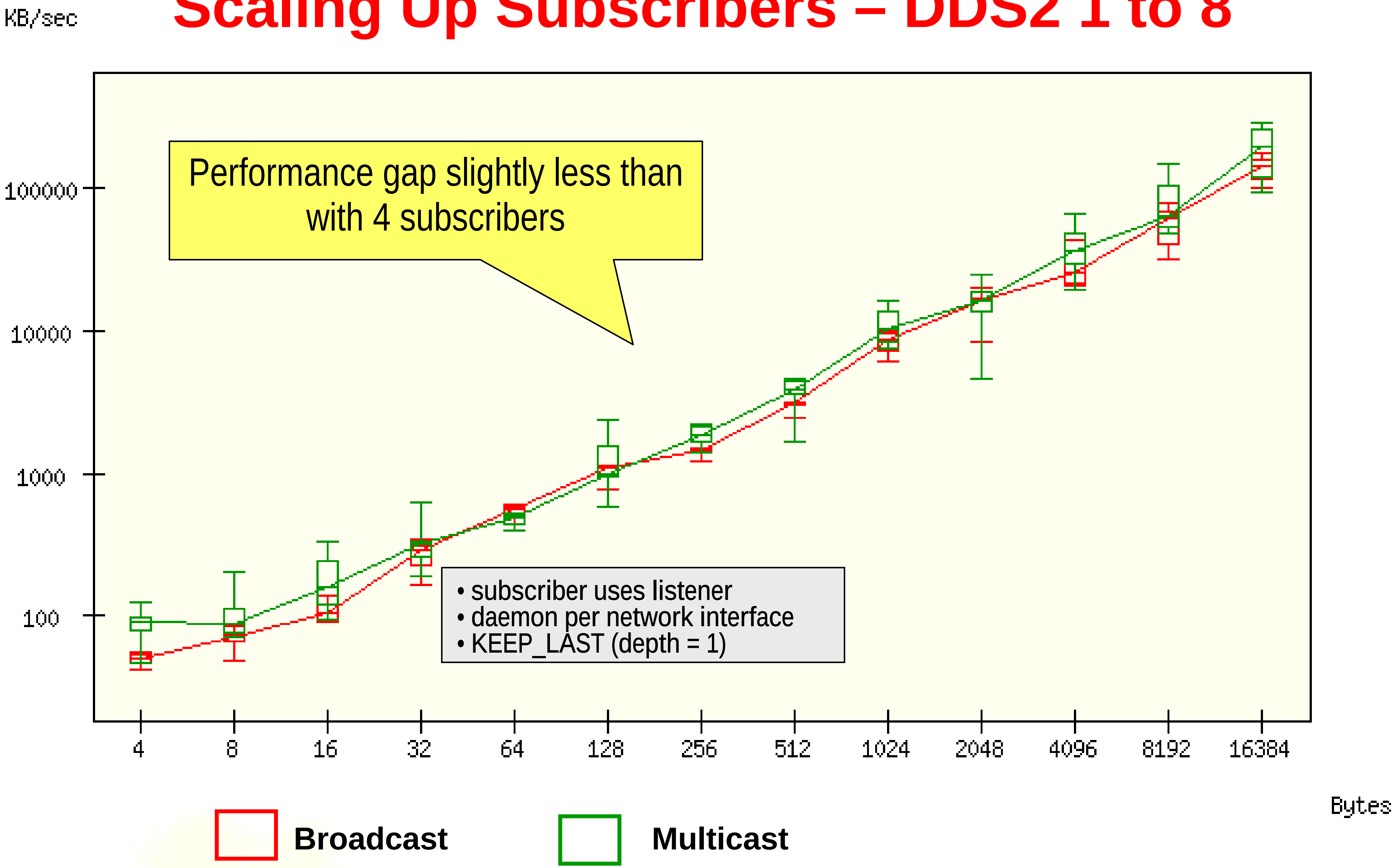
Scaling Up Subscribers – DDS2 Multicast



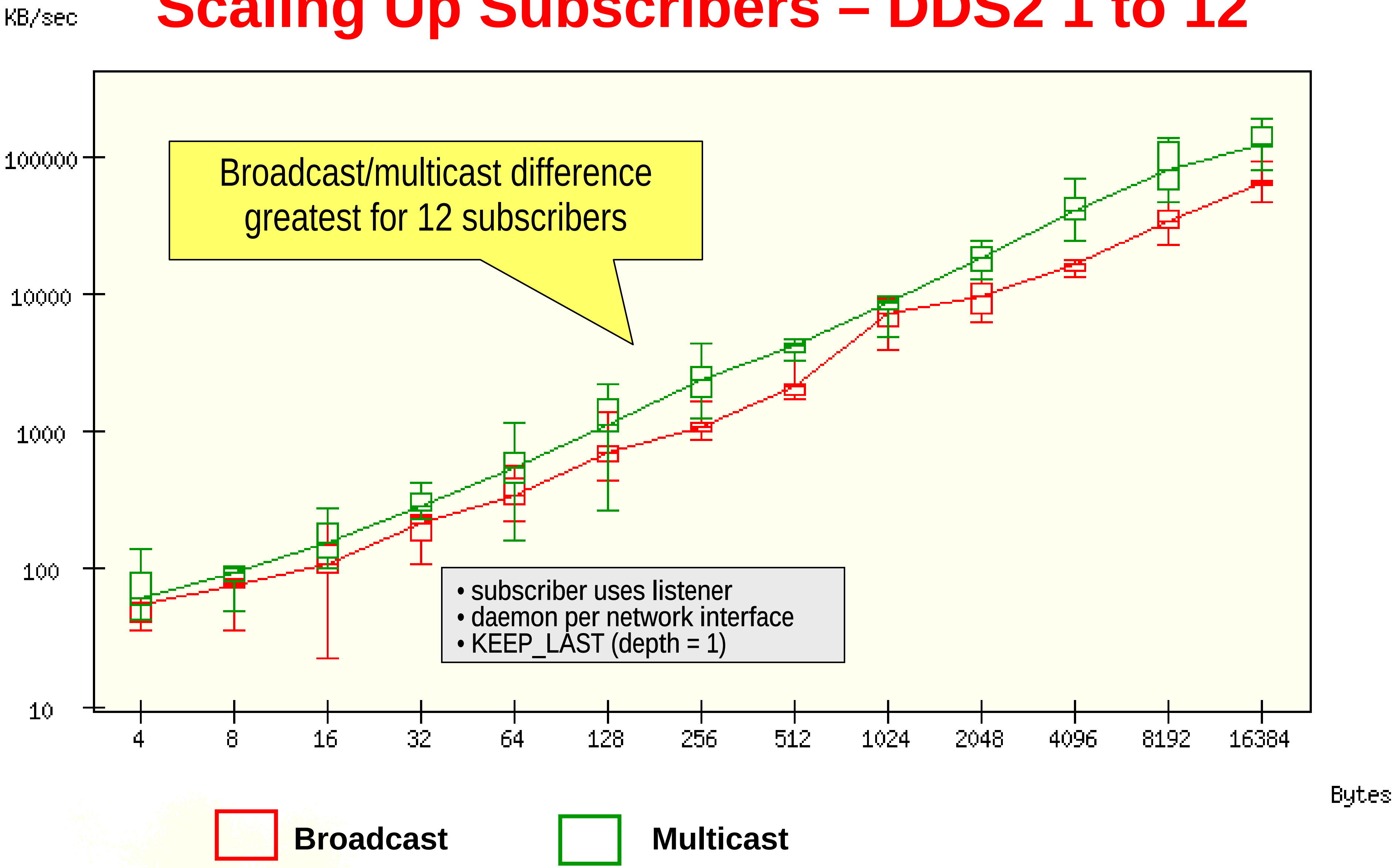
Scaling Up Subscribers – DDS2 1 to 4



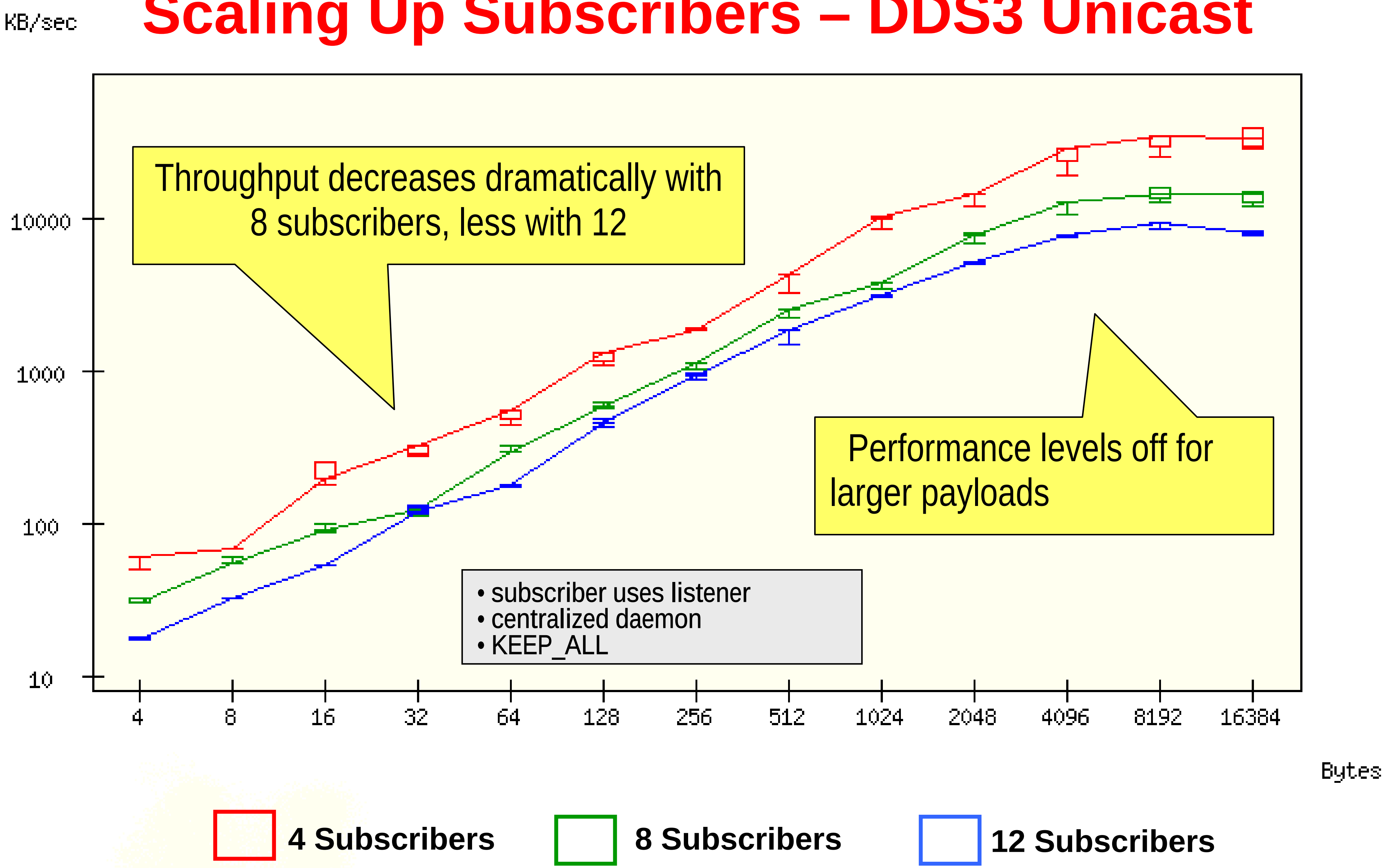
Scaling Up Subscribers – DDS2 1 to 8



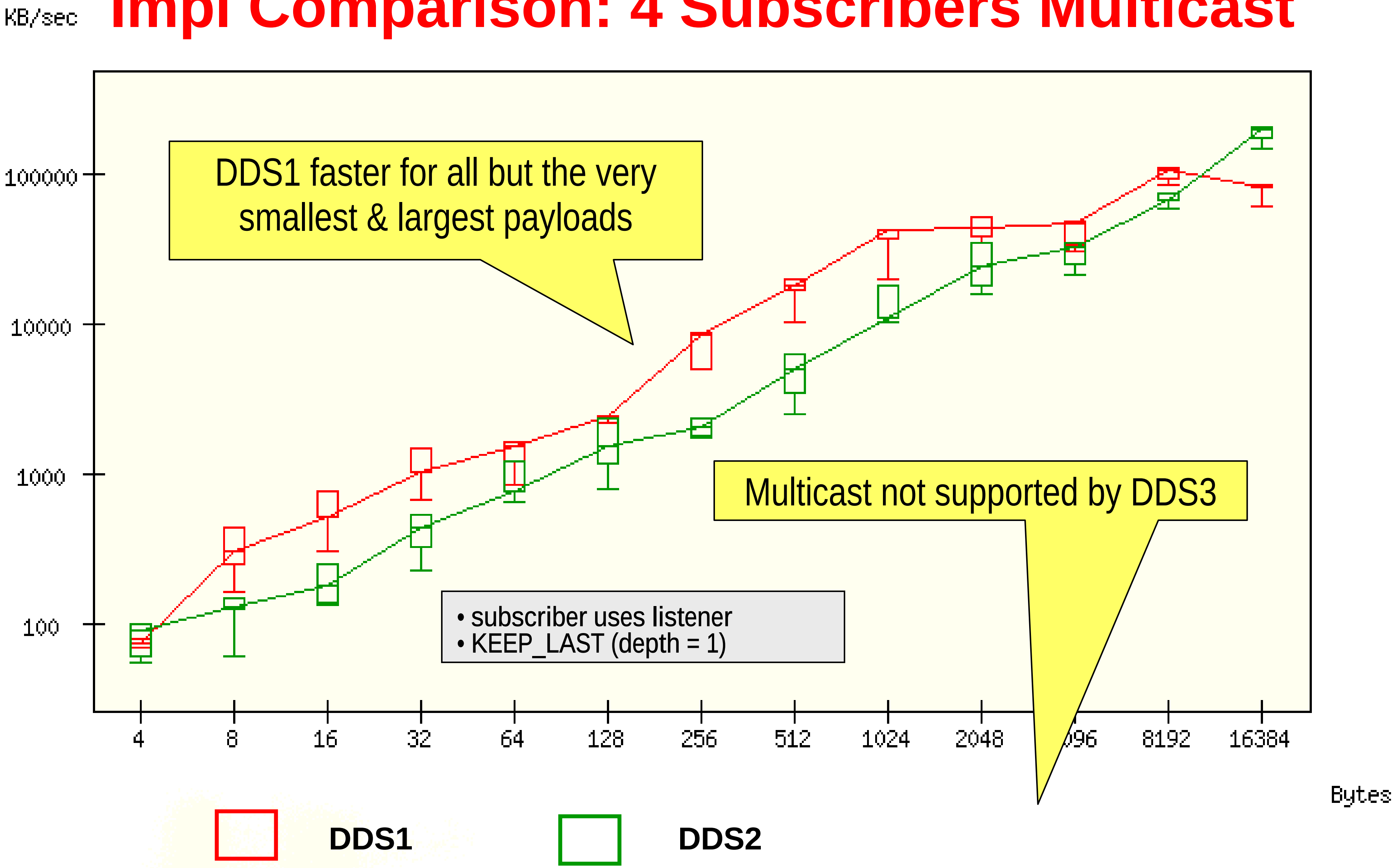
Scaling Up Subscribers – DDS2 1 to 12



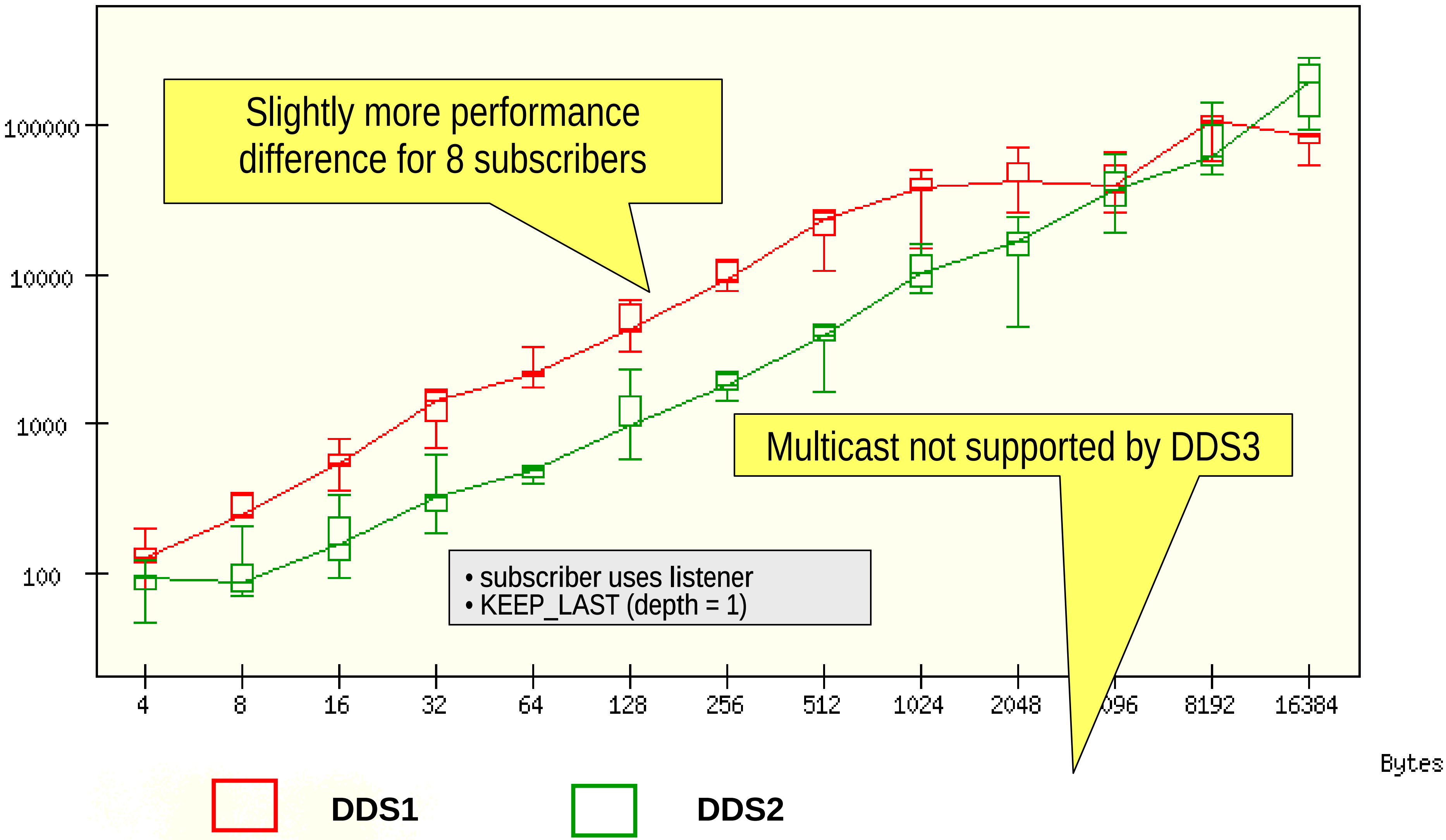
Scaling Up Subscribers – DDS3 Unicast



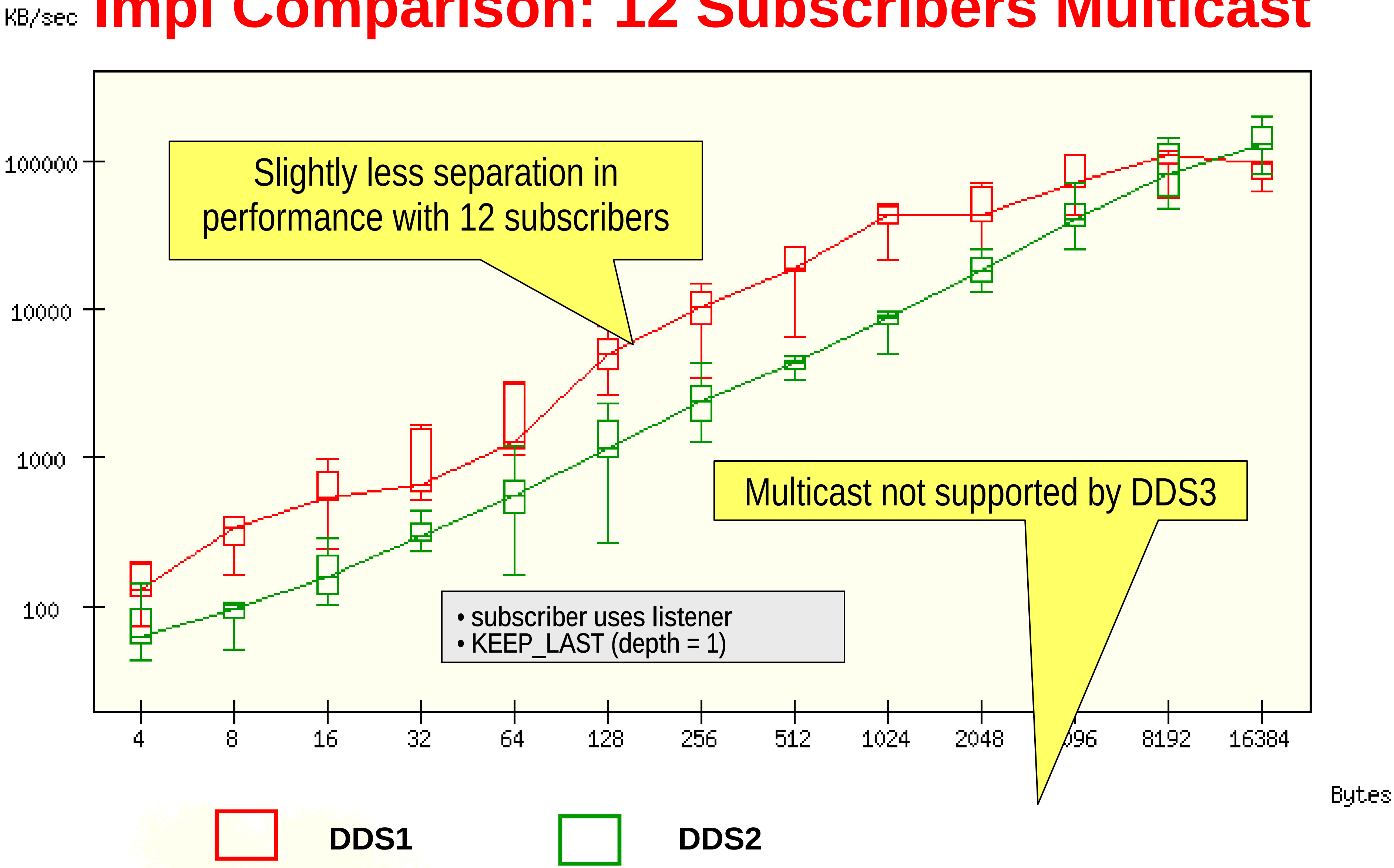
Impl Comparison: 4 Subscribers Multicast



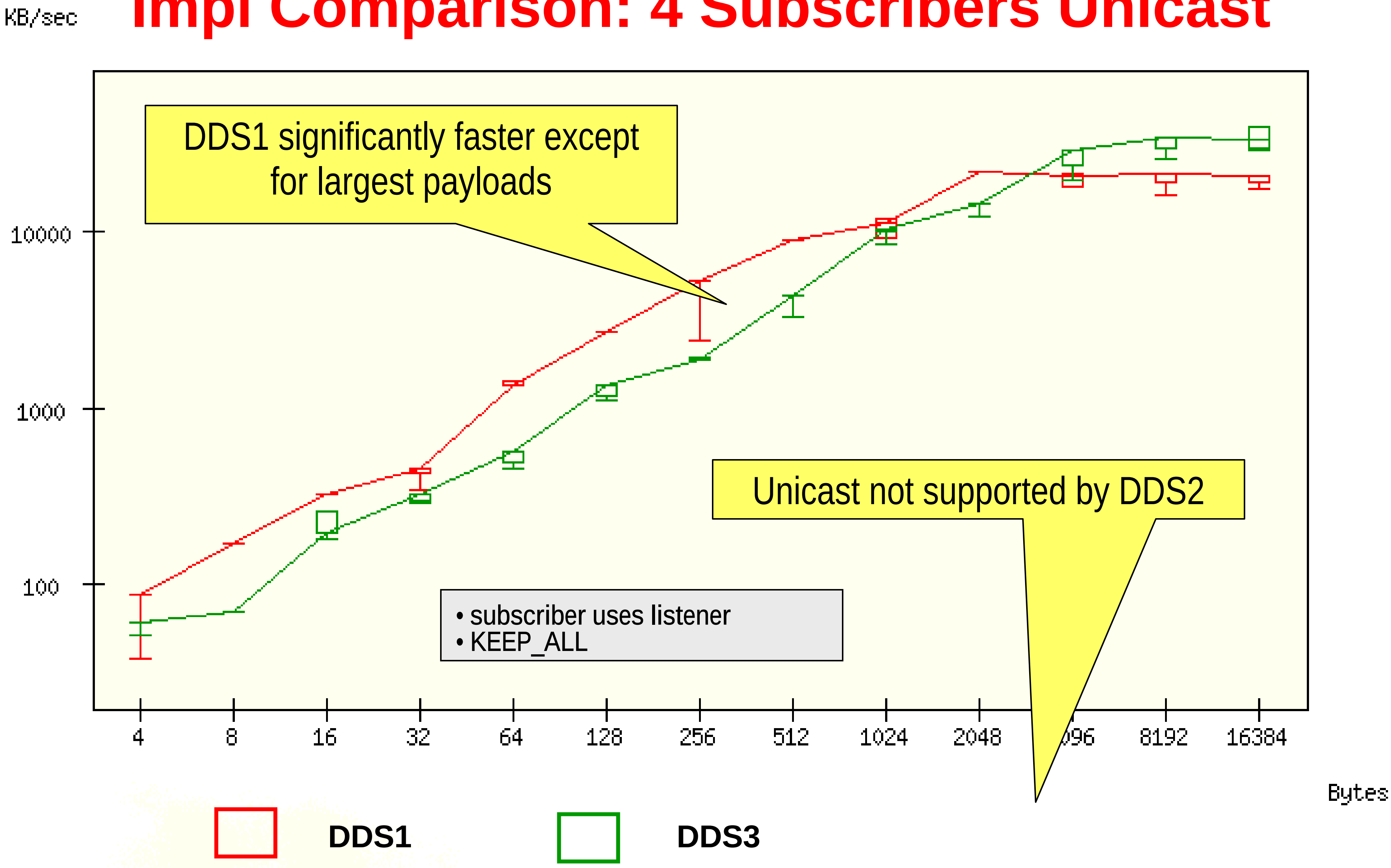
Impl Comparison: 8 Subscribers Multicast



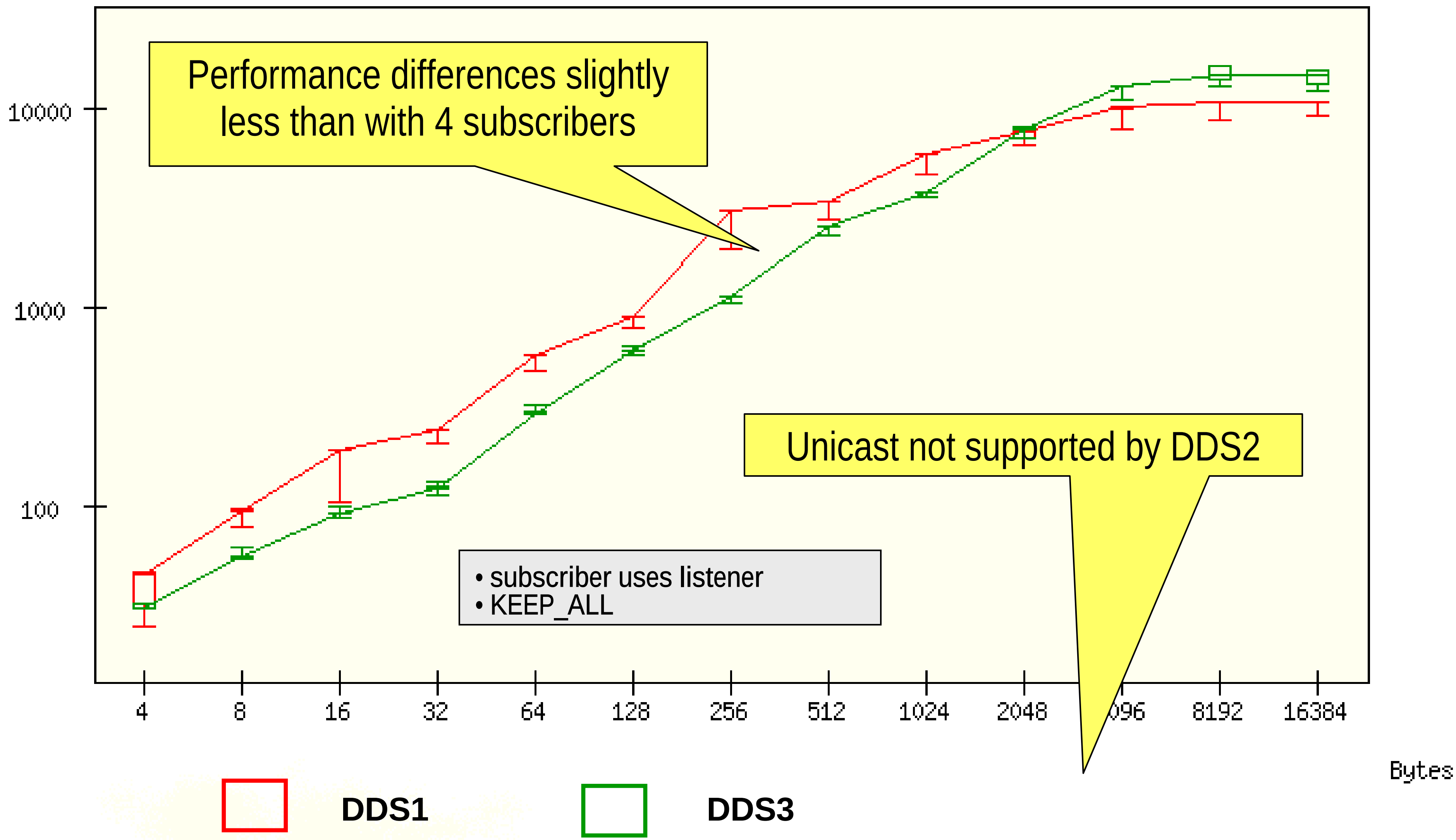
Impl Comparison: 12 Subscribers Multicast



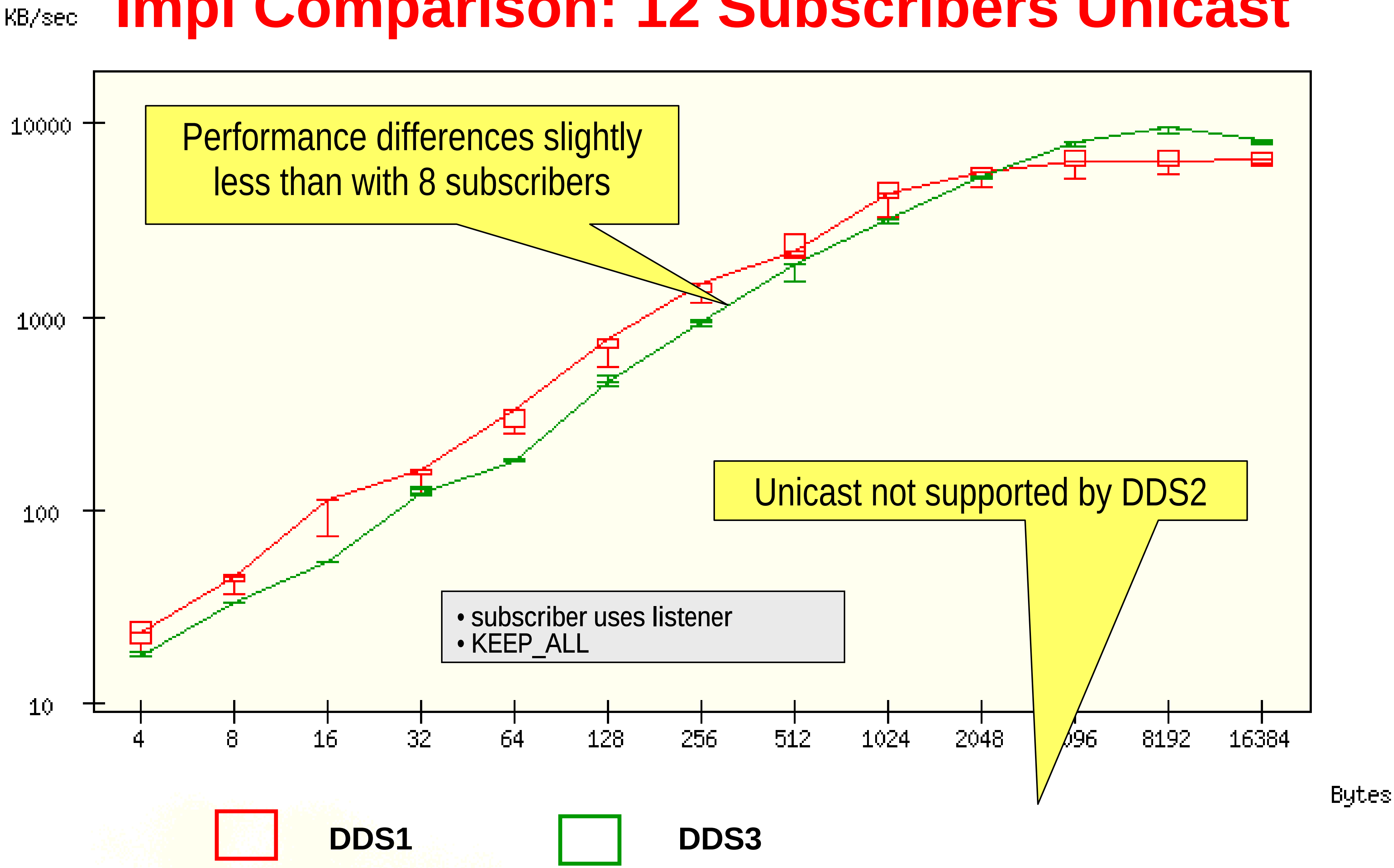
Impl Comparison: 4 Subscribers Unicast



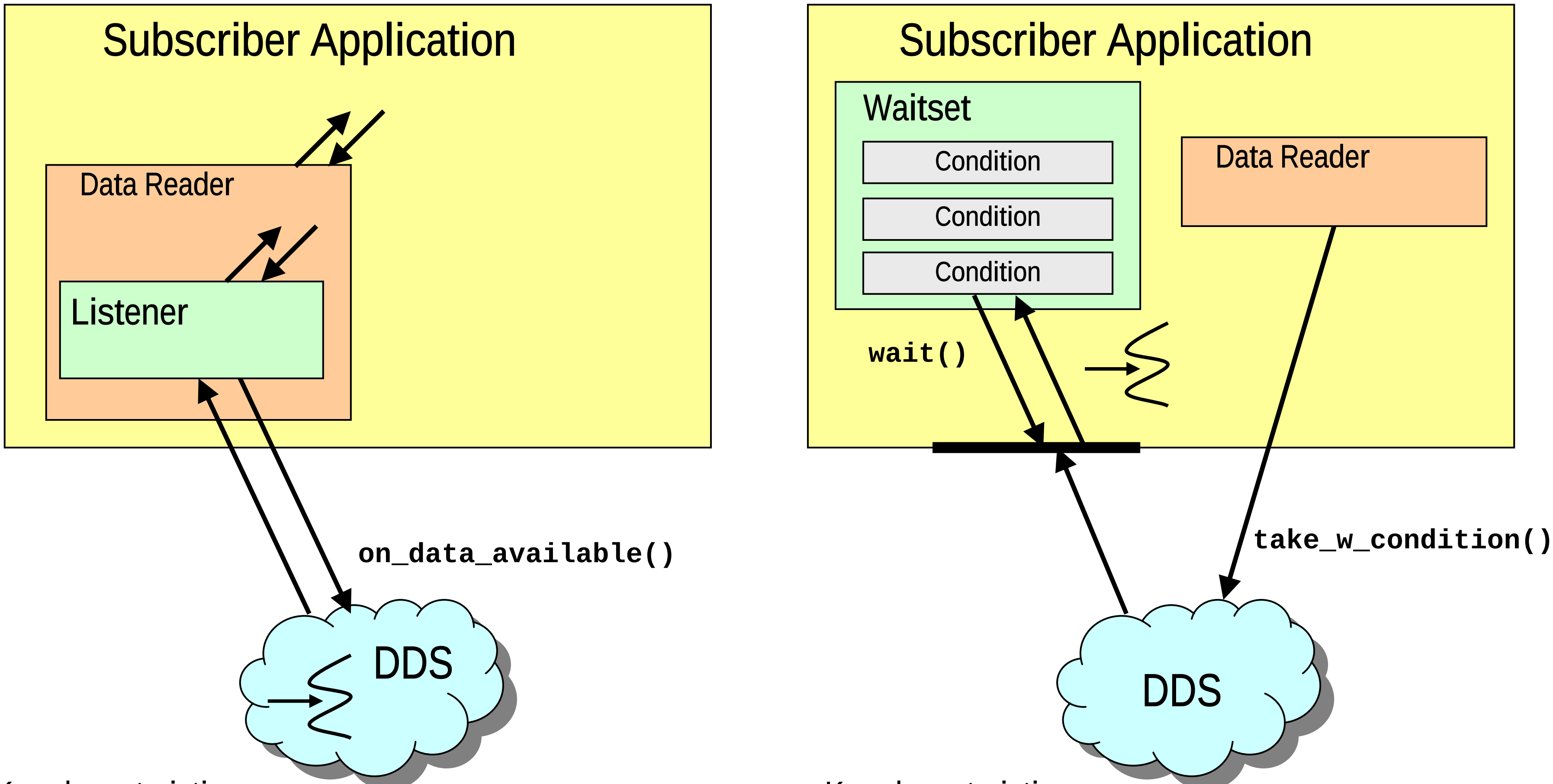
Impl Comparison: 8 Subscribers Unicast



Impl Comparison: 12 Subscribers Unicast



Overview of DDS Listener vs. Waitset



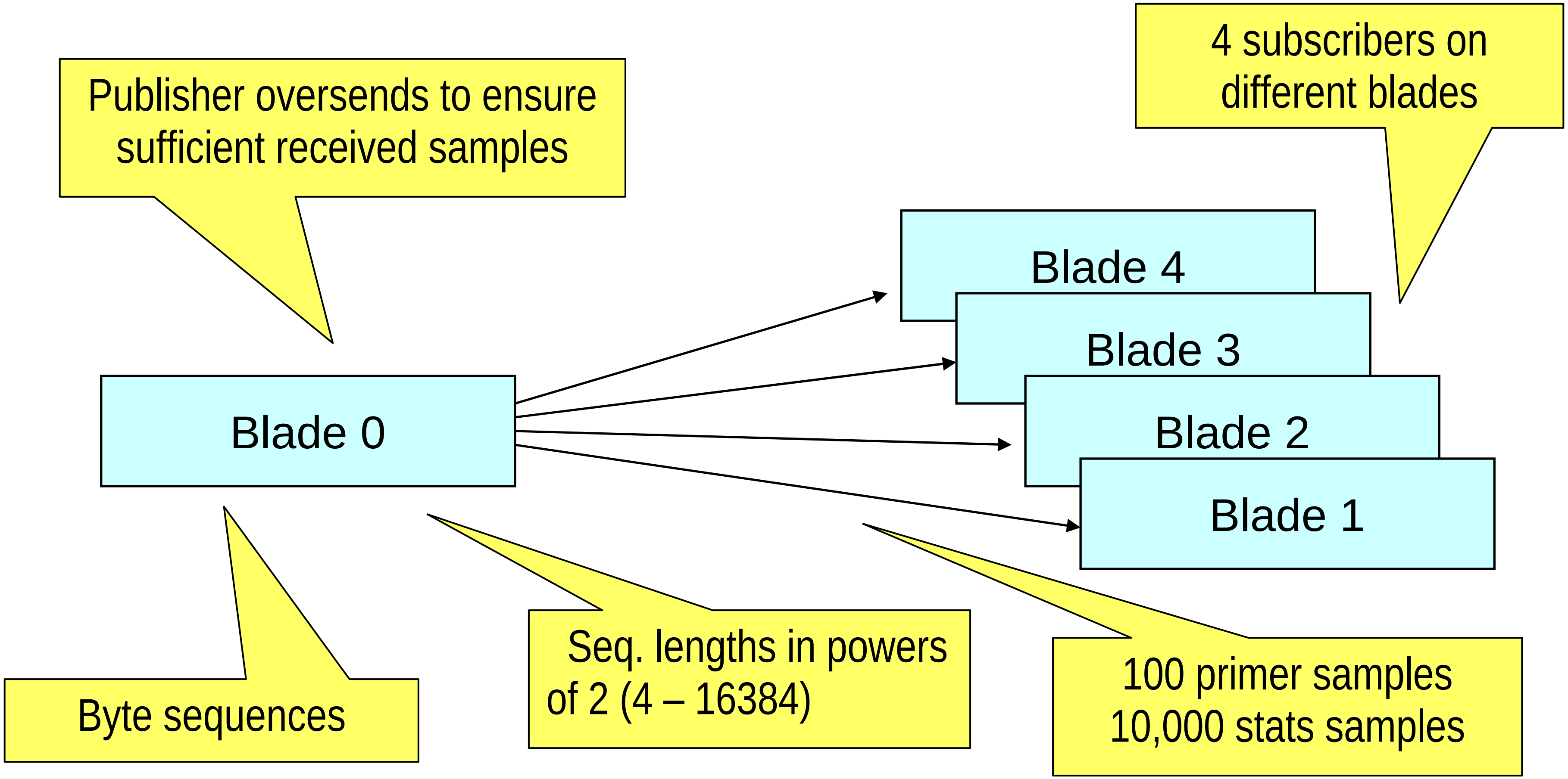
Key characteristics

- No application blocking
- DDS thread executes application code

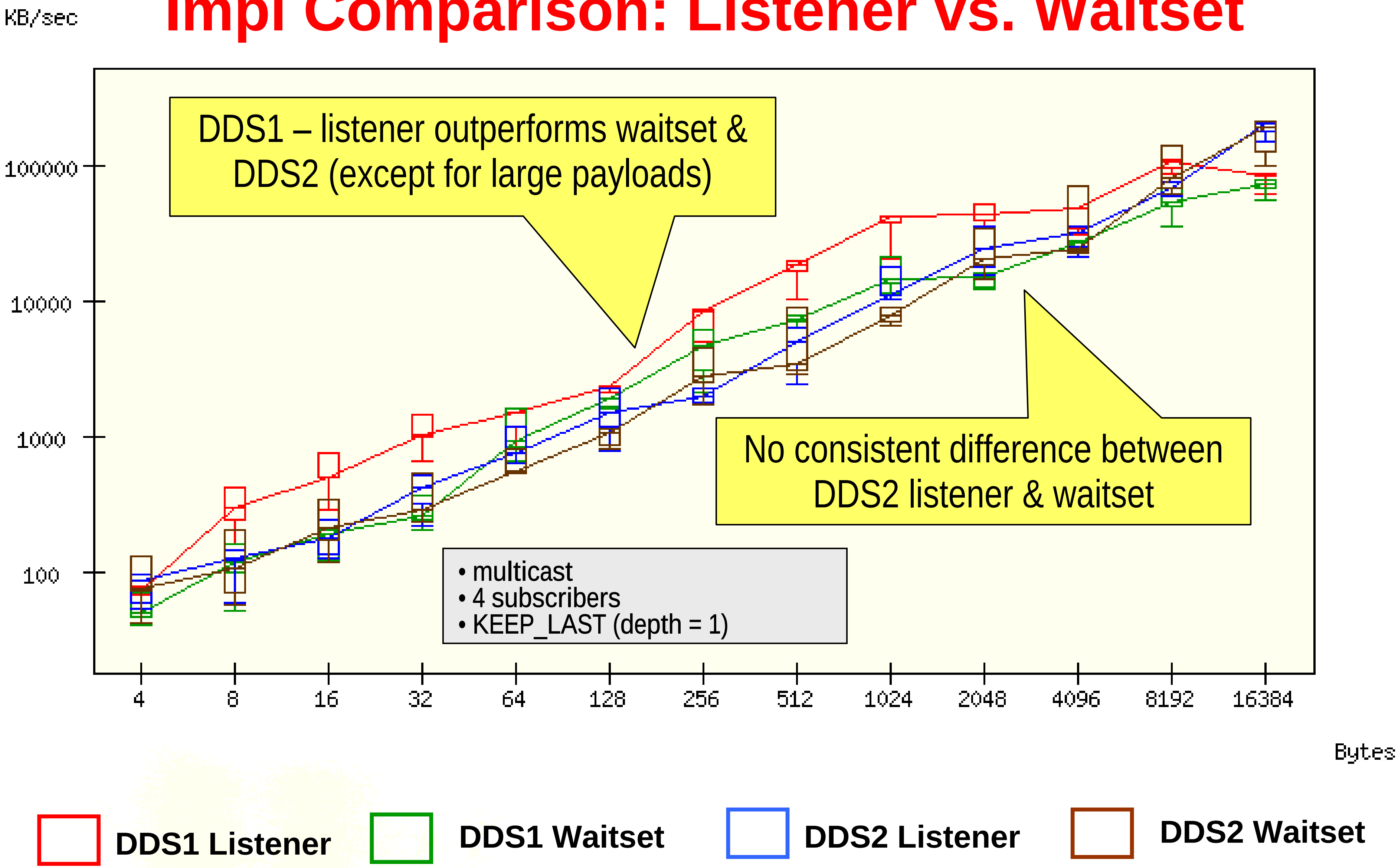
Key characteristics

- Application blocking
- Application has full control over priority, etc.

Comparing Listener vs Waitset Throughput

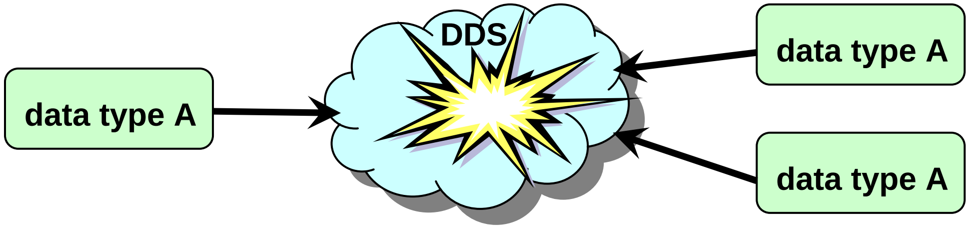


Impl Comparison: Listener vs. Waitset



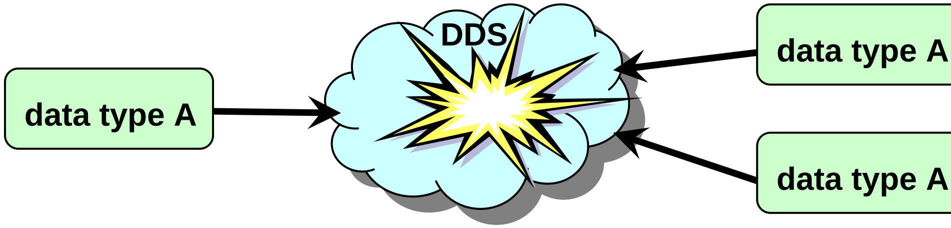
DDS Application Challenges

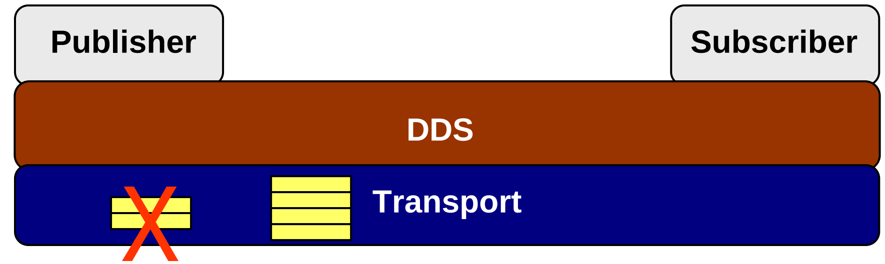
- Scaling up number of subscribers
  - Data type registration race condition (DDS3)
  - Setting proprietary ‘participant index’ QoS (DDS1)



DDS Application Challenges

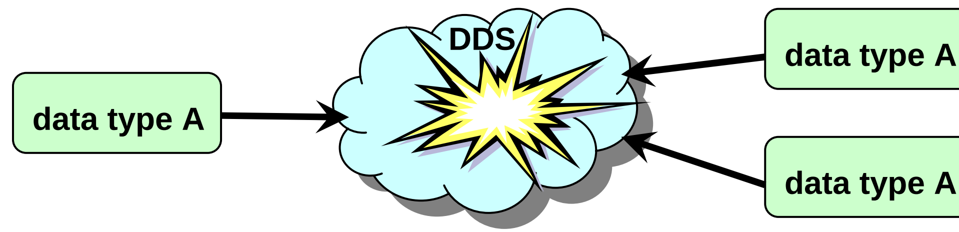
- Scaling up number of subscribers
  - Data type registration race condition (DDS3)
  - Setting proprietary ‘participant index’ QoS (DDS1)
- Getting a sufficient transport buffer size

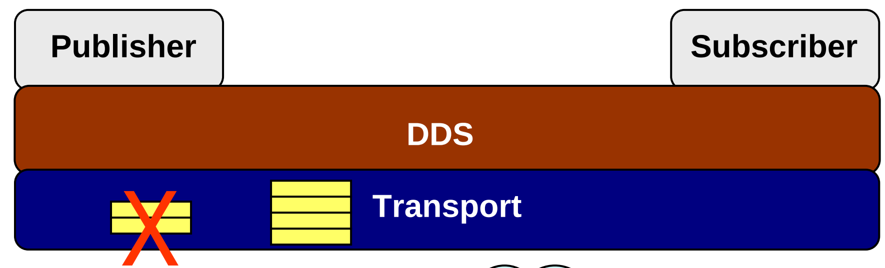


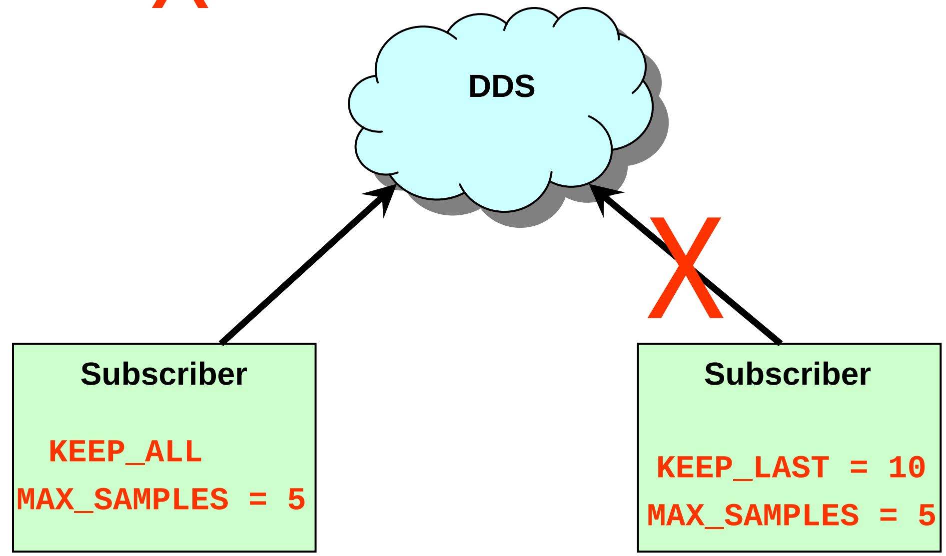


DDS Application Challenges

- 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		member method	member method
Spec Operations	extra argument (newer spec)	compliant	compliant
Key Declaration		sin[illegible]	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
[illegible]	extra ar... [illegible]	...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	// [illegible]	single [illegible]	[illegible] ir of [illegible]
R [illegible]	pu [illegible]	none	publisher
[illegible] Transport Config	[illegible]	none	file-based or code-based

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

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

Portability Challenges

	DDS1	DDS2	DDS3
Do… 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
	publi… subs…		…lisher
	code-based	none	…e-based or code-based

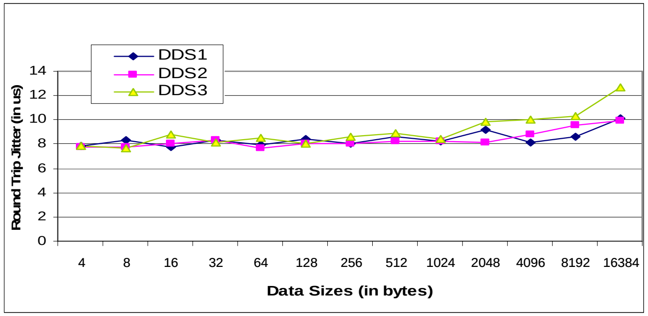
```
#pragma keylist Info id
```

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

```
#pragma DCPS_DATA_TYPE "Info"
#pragma DCPS_DATA_KEY "id"
```

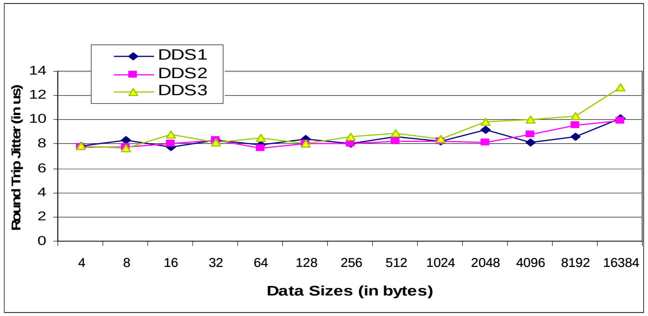
Lessons Learned - Pros

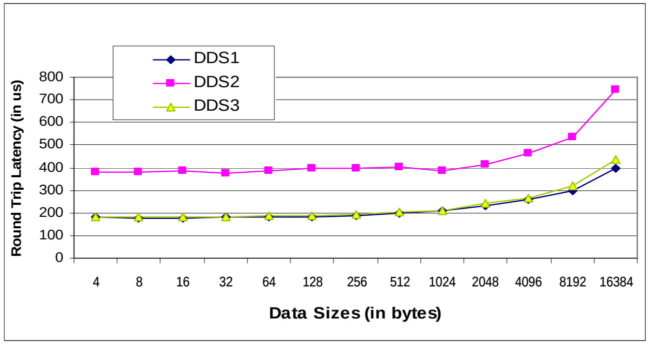
- DDS implementations are optimized for different use cases & design spaces
  - Low latency for collocated publishers and subscribers



Lessons Learned - Pros

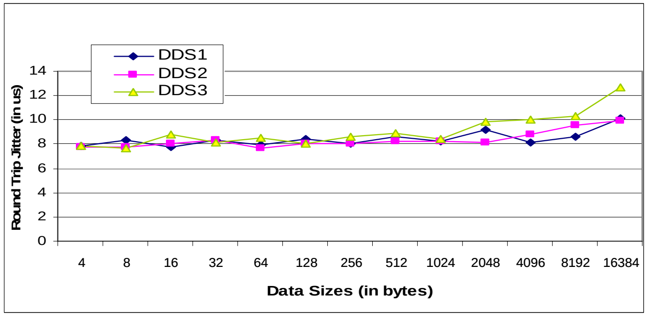
- DDS implementations are optimized for different use cases & design spaces
  - Low latency for collocated publishers and subscribers
  - Low latency for remote publishers and subscribers

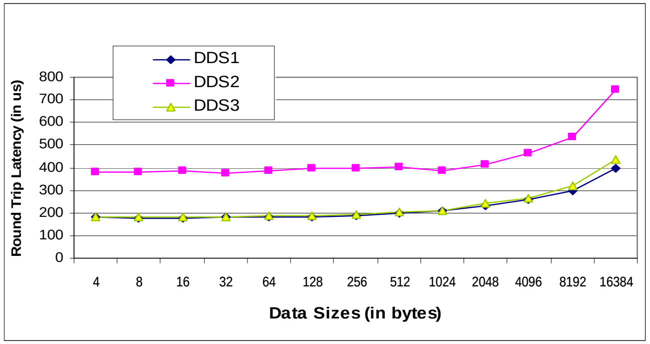


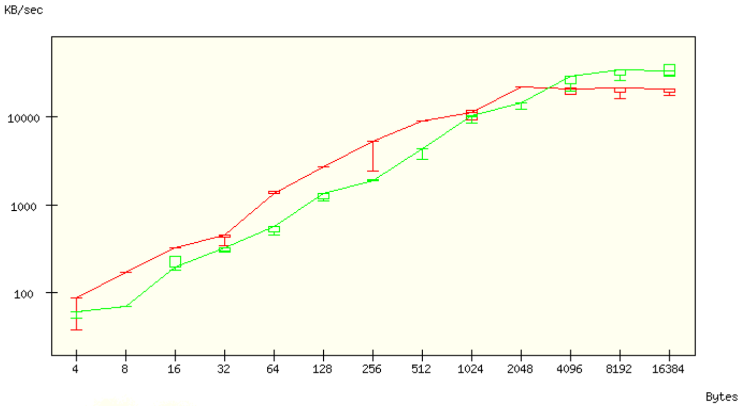


Lessons Learned - Pros

- 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

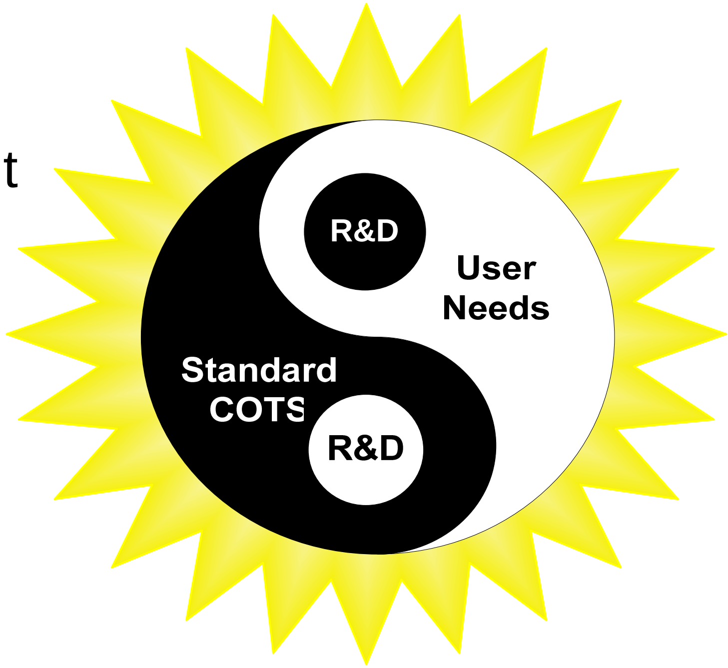
- Alternate throughput measurement strategies
  - Fixed # of samples – measure elapsed time
  - Fixed time window – measure # of samples
  - Controlled publish rate
- Generic testing framework
  - Common test code
  - Wrapper facades to factor out portability issues
- Include other pub/sub platforms
  - 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