Middleware Options for the Keck Common Services Framework

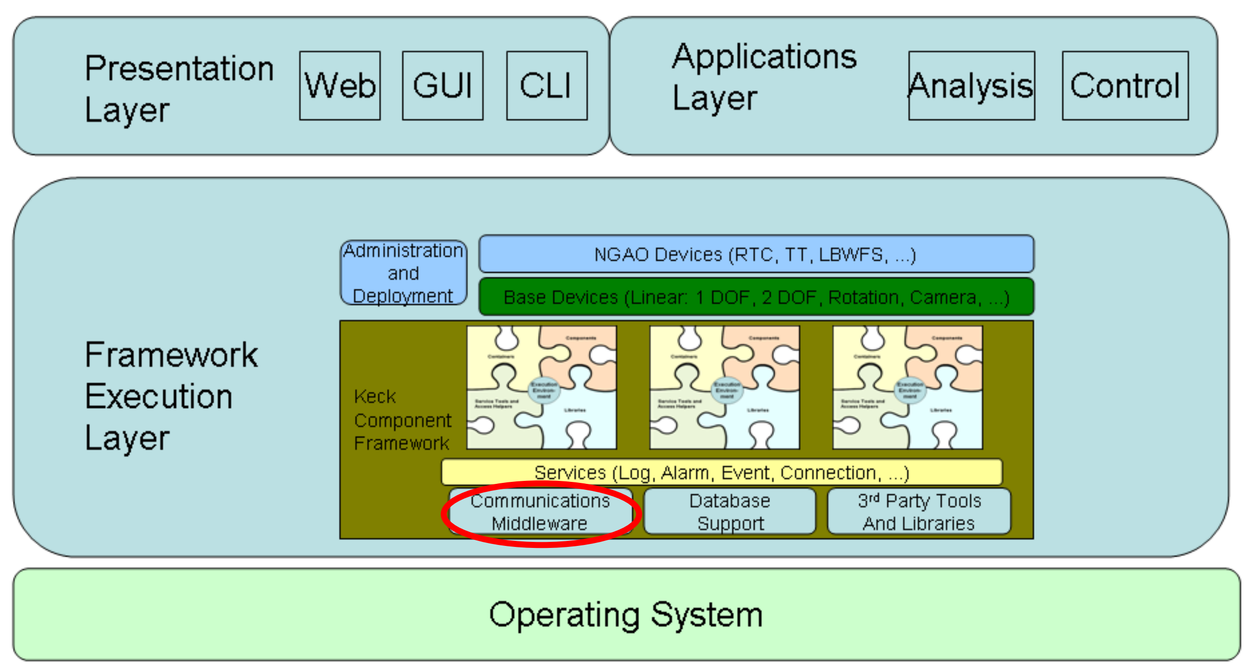
NGAO Control SW Team:
Erik Johansson, Jimmy Johnson, Doug Morrison
August 14, 2009

What is Middleware?

- Middleware is communication software that allows software components to communicate with each other without regard to their physical location on the network.
- Examples of middleware:
  - Channel Access (EPICS)
  - Common Object Request Broker Architecture (CORBA)
  - Internet Communication Engine (ICE)
  - Data Distribution Service (DDS)
  - Distributed Component Object Module (DCOM)

How is middleware used in KCSF?

- KCSF uses middleware to communication between components:
  - Peer to peer command processing
  - Publish-subscribe communications
- The middleware is encapsulated so the user sees only a simple interface and knows nothing about the underlying implementation.
- We are considering both ICE and DDS for use with KCSF.

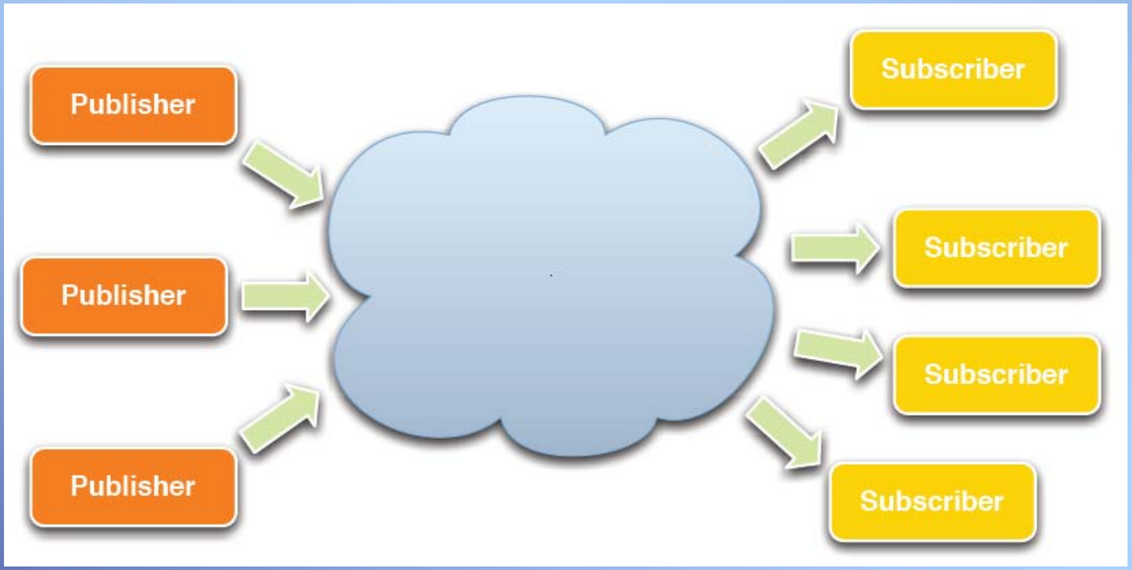


Data Distribution Service (DDS)

- An OMG standard based on a publish-subscribe paradigm
- Widely used in the DoD for distributed network applications
- Available from multiple vendors in both open source and commercial versions. Two main vendors:
  - Real-Time Innovations (RTI). Commercial versions only.
  - PrismTech. Both open source and commercial versions
- Multiple language support: C, C++, Java
- Multiple platform/OS support: Windows, Linux, Solaris, VxWorks
- High throughput, low latency, low jitter
- Tunable “Quality of Service” parameters
- Supports multi-cast communication

How does publish-subscribe work?

- Communication via “Topics”
- Topics identified by unique name
- One or more publishers per topic
- Zero or more subscribers per topic
- Publishers and subscribers are unaware of each other
- Publishers and subscribers can be located anywhere on the network

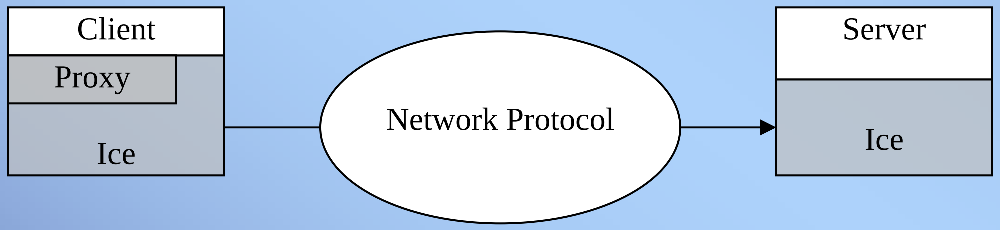


Internet Communication Engine (ICE)

- Object oriented middleware for building distributed applications.
- Successor to CORBA. Faster and more efficient.
- Many of the original CORBA designers are now with ZeroC.
- Based on a client-server model.
- Publish-subscribe communications available through IceStorm.
- Only a single vendor: ZeroC, Inc.
- Available under open-source or commercial license.
- Large customer base, including Lockheed Martin, SGI, Northrop Grumman and HP.
- Multiple language support: C++, Java, Ruby, Python
- Multiple platform/OS support: Windows, Linux, Solaris, Mac OSX (no VxWorks yet).
- Reliable transport mechanism.

Client-server model

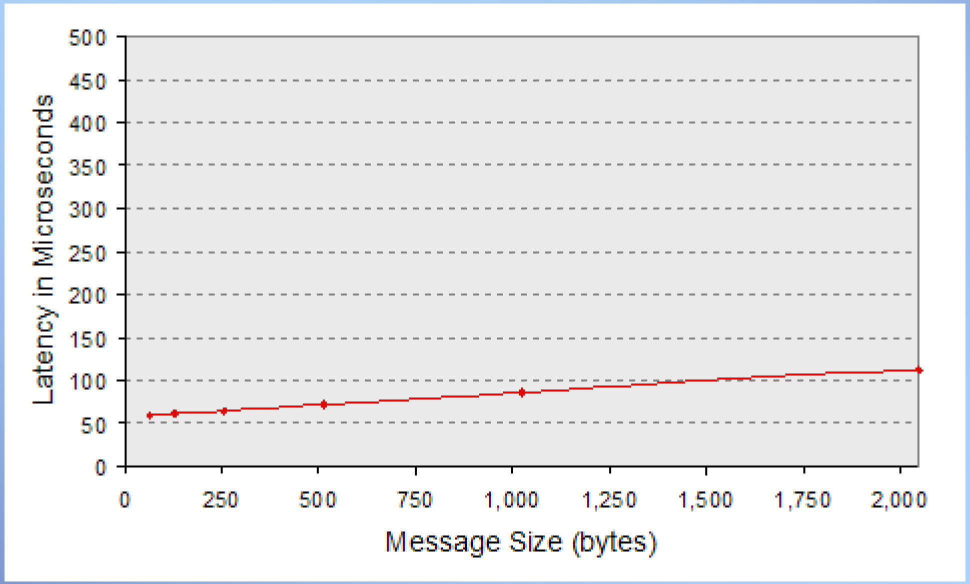
- Client must connect directly to server.
- Client receives a proxy through which it can execute methods on the remote object.
- Connections between components are tightly coupled.
- Good for command-response environments.



RTI DDS: Performance

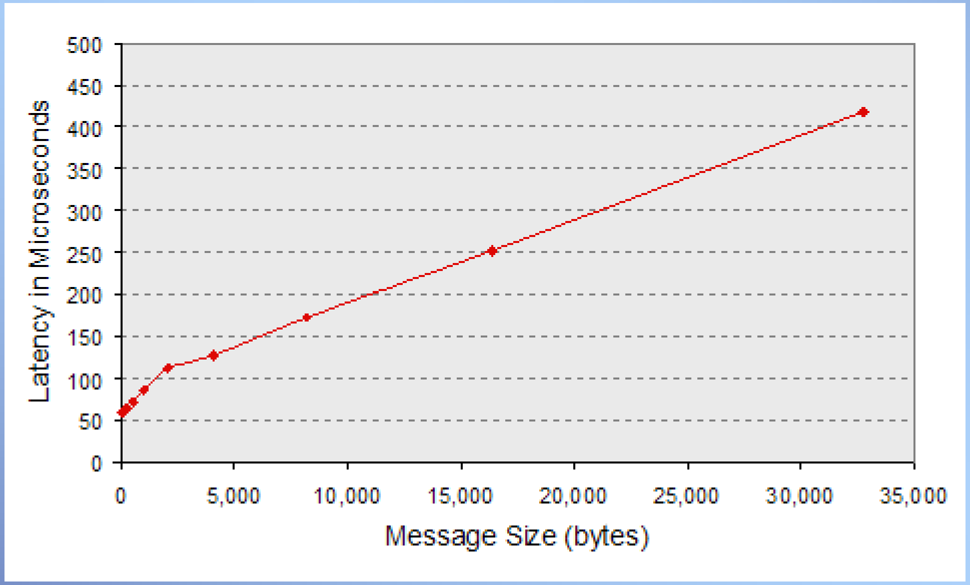
- Test environment:
  - RTI Data Distribution Service 4.3
  - Red Hat Enterprise Linux 5.0, 32-bit
  - 2.4 GHz processors – mix of Intel Core 2 Duo E6600 and Core 2 Quad Q6600
  - Gigabit Ethernet
  - Intel PRO/1000 NIC
  - D-Link DGS-3324SRi switch
  - UDP over IPv4
  - Reliable messaging with ordered delivery

RTI DDS: Latency (one-way)



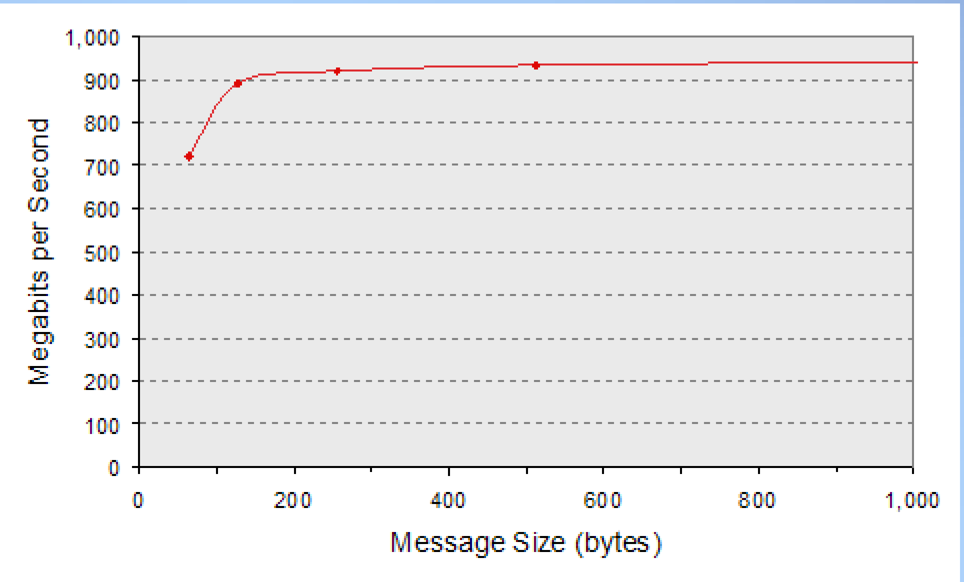
Small Messages

Large Messages

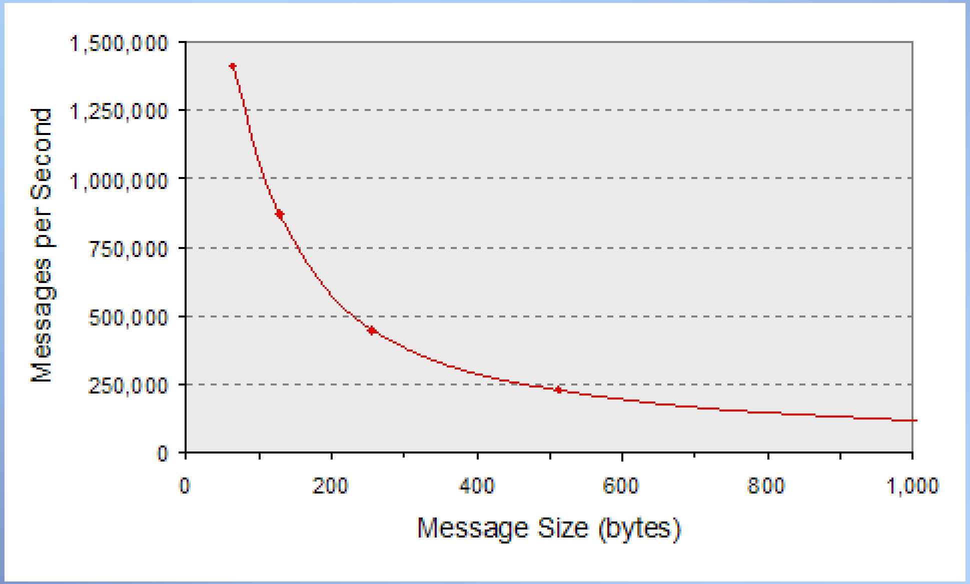


RTI DDS: One-to-one throughput

- For messages larger than 128 bytes, throughput is limited by network, not DDS

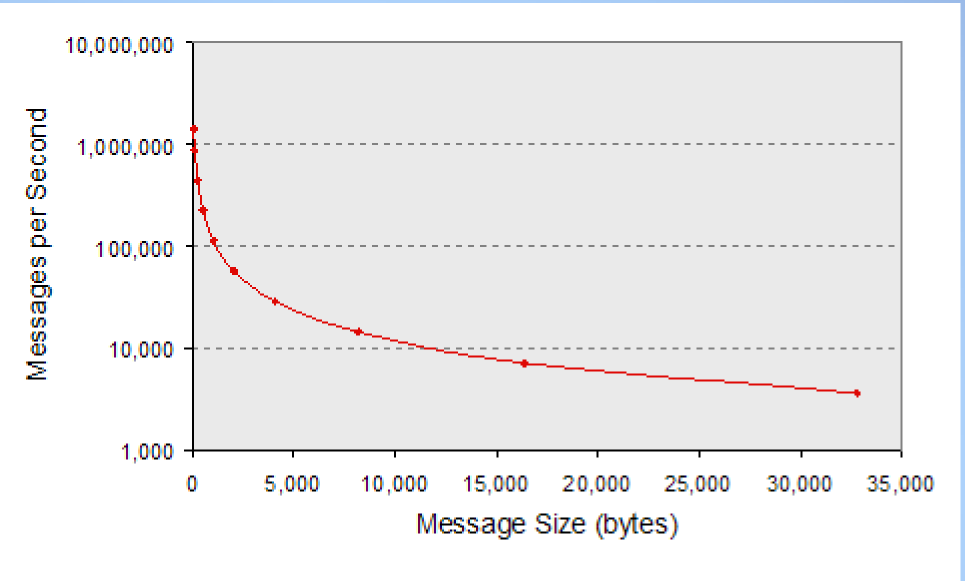


RTI DDS: One-to-one message rate



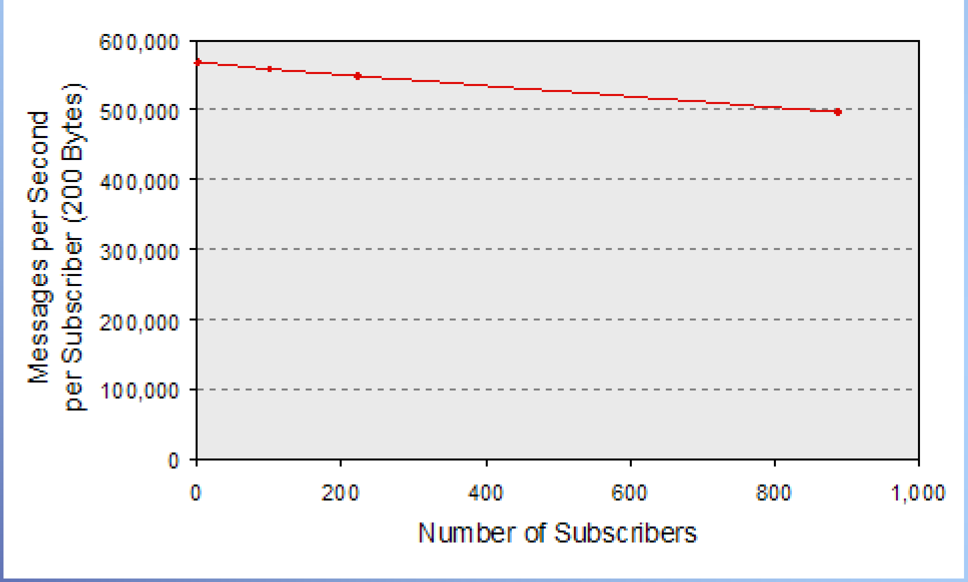
Small Messages

Large Messages



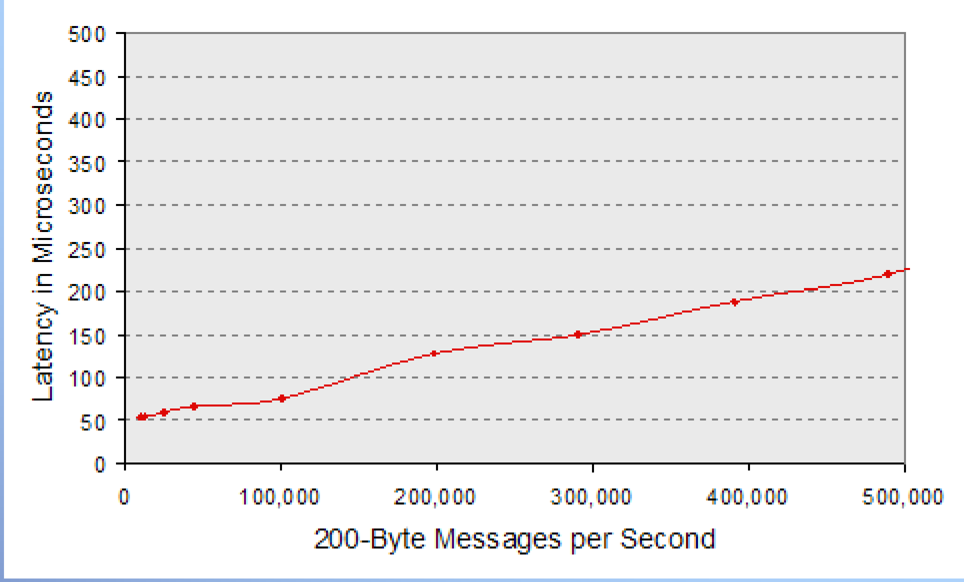
RTI DDS: One-to-many throughput scalability

- Number of subscribers has only small effect on throughput
- Single thread used to send a stream of 200 byte messages to up to 888 subscribers
- Each subscriber running on a dedicated core, 4 subscribers per network interface.
- Intel Xeon processors were used for this benchmark



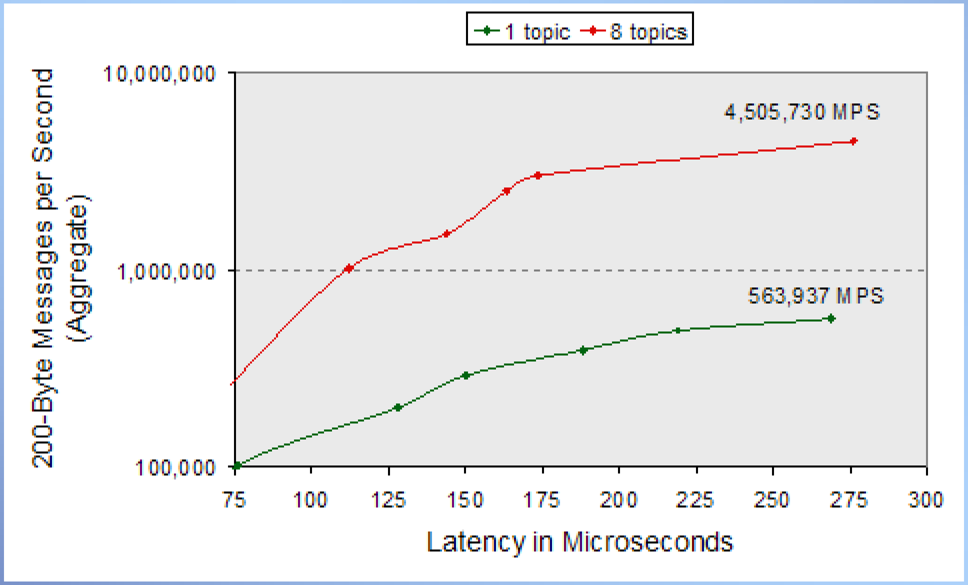
RTI DDS: Impact of throughput on latency

- Latency remains low even near network saturation conditions.



RTI DDS: Topic and capacity scalability

- Data points are from previous plot on latency and throughput
- Green points: single topic, Red points: 8 topics
- Capacity scales proportionately with number of topics



ICE Performance

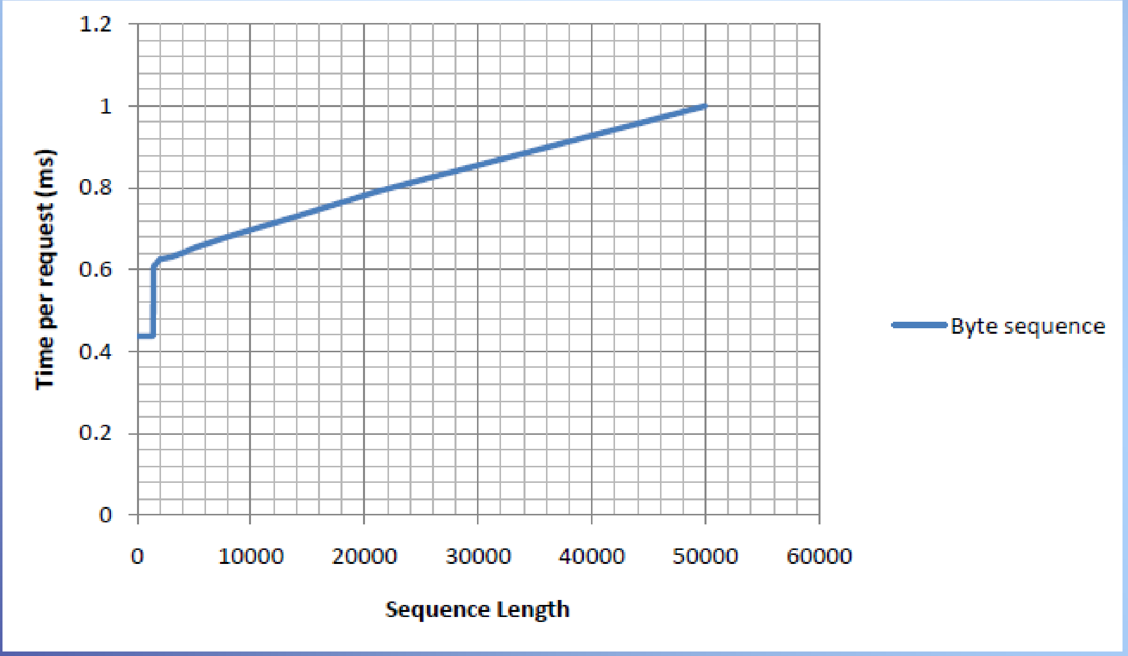
- Test environment:
  - Dual-core 2.2GHz Athlon, 2GB RAM, Win XP Pro, SP3
  - Dual-core 2.0 GHz Mac Mini, 2GB RAM, Win Vista Ultimate SP1
  - C++, C#: Visual Studio 2008
  - Java: 1.60 JDK
  - Ice version 3.3.0
  - Code optimized for speed.
  - Used 64-bit code on Vista machine
  - Loopback tests done on Athlon
  - Network tests: Client → Athlon, Server → Mac Mini

ICE Performance, Latency

Latency (round-trip), no data sent

Requests/second	Ice for .NET	Ice for Java	Ice for C++
Loopback	6,900	8,000	10,500
Gigabit network	2,300	2,300	2,300

Latency as a function of message size



ICE Performance, Throughput

Throughput, loopback:

Throughput (loopback)	Ice for .NET	Ice for Java	Ice for C++
Byte seq (send)	630Mbit/s	800Mbit/s	1,200Mbit/s
Byte seq (recv)	610Mbit/s	720Mbit/s	960Mbit/s
Fixed seq (send)	380Mbit/s	140Mbit/s	620Mbit/s
Fixed seq (recv)	300Mbit/s	110Mbit/s	530Mbit/s
Variable seq (send)	68Mbit/s	65Mbit/s	190Mbit/s
Variable seq (recv)	62Mbit/s	70Mbit/s	150Mbit/s

Throughput, gigabit network:

Throughput (gigabit network)	Ice for .NET	Ice for Java	Ice for C++
Byte seq (send)	520Mbit/s	660Mbit/s	740Mbit/s
Byte seq (recv)	410Mbit/s	590Mbit/s	655Mbit/s
Fixed seq (send)	300Mbit/s	250Mbit/s	525Mbit/s
Fixed seq (recv)	205Mbit/s	150Mbit/s	470Mbit/s
Variable seq (send)	55Mbit/s	180Mbit/s	255Mbit/s
Variable seq (recv)	55Mbit/s	75Mbit/s	145Mbit/s