QCon London

6.th International
Software Development
Conference 2012

Training : March 5 - 6 // Conference : March 7 - 9

High Performance Network Applications in the Capital Markets

Todd L. Montgomery
VP Architecture, Messaging Business Unit
@toddlmontgomery

INFORMATICA®

Why do Developers use Messaging?

Message-Oriented Middleware (MOM)

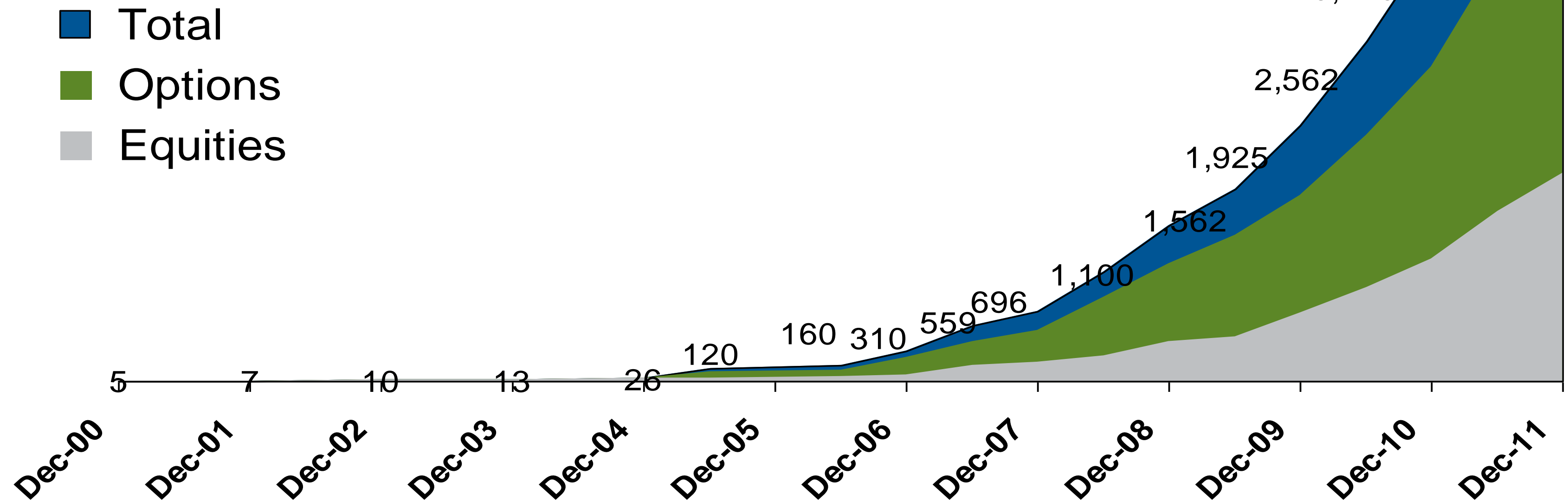
- **Abstraction (Pub-Sub, Req/Resp, Queuing)**
  - Separate physical systems from communication
  - Easily modify logic and scale applications
- **Functionality**
  - Guaranteed delivery, fault tolerance, load balancing…
- **Efficiency**
  - Well designed messaging systems reduce infrastructure
- **Leverage broad, deep and detailed expertise**
  - Focus on core competencies, Faster Time-to-Market

Market Data Growth

Data Deluge

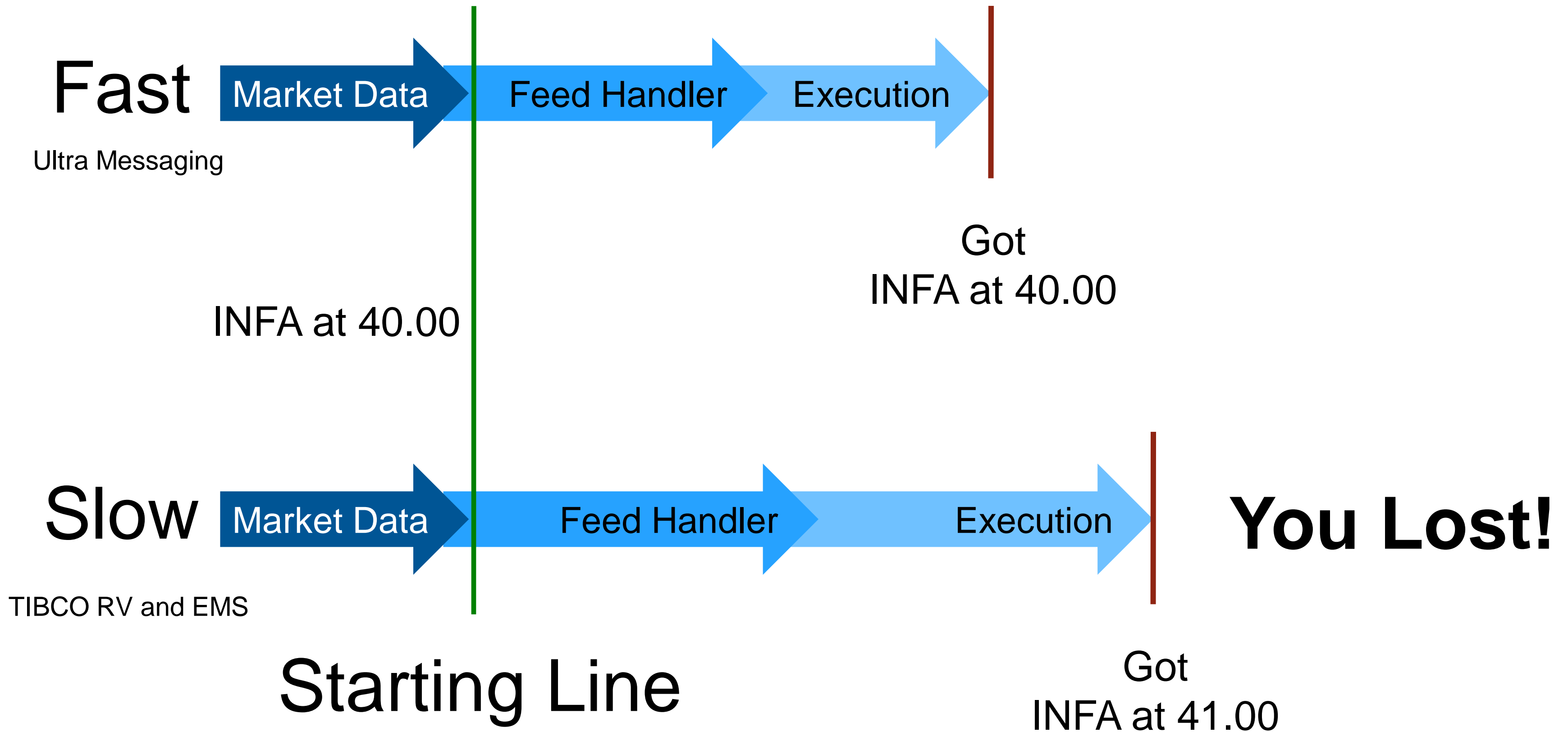
Aggregated One Minute
Peak ***Messages Per Second*** Rates
Arca, CTS, CQS, OPRA, NQDS
(in *thousands*)

> 1Terabyte of Data per Day



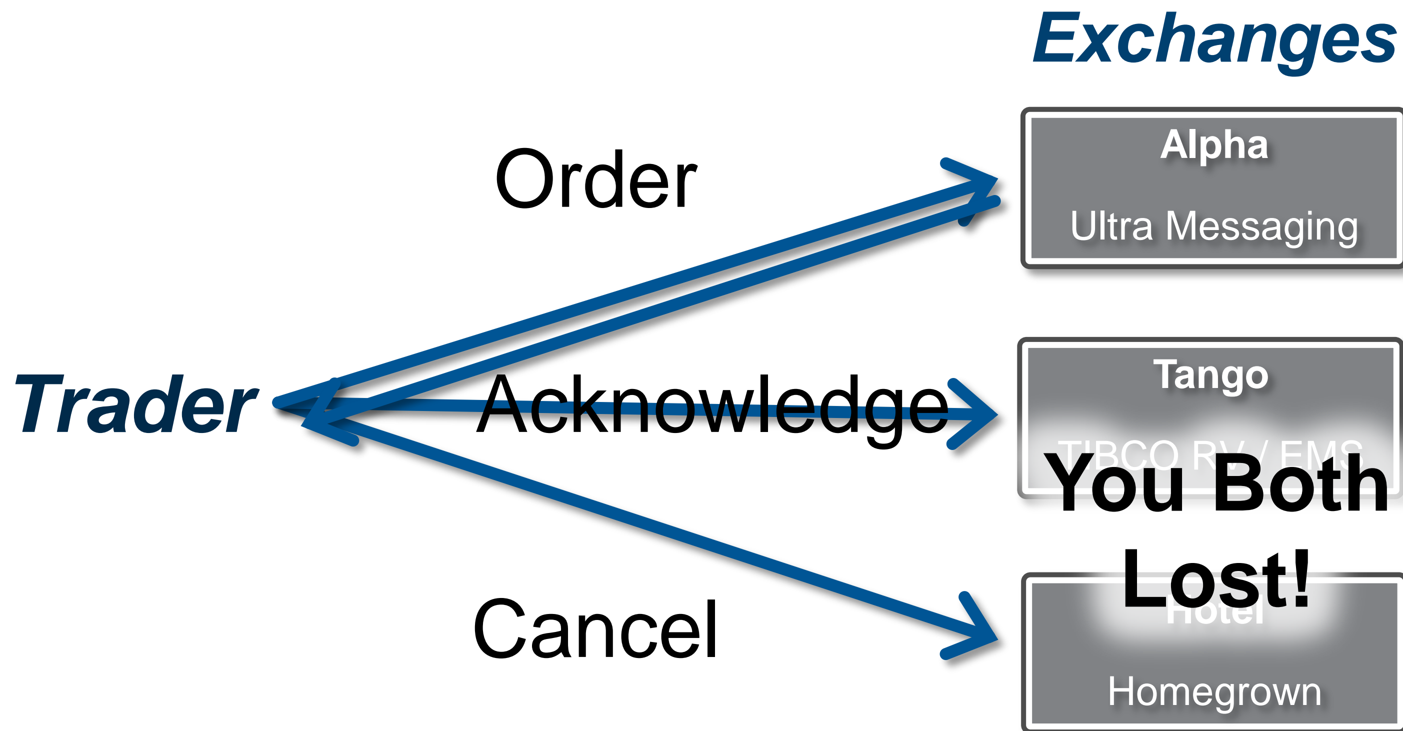
The Trader

Why Latency Matters



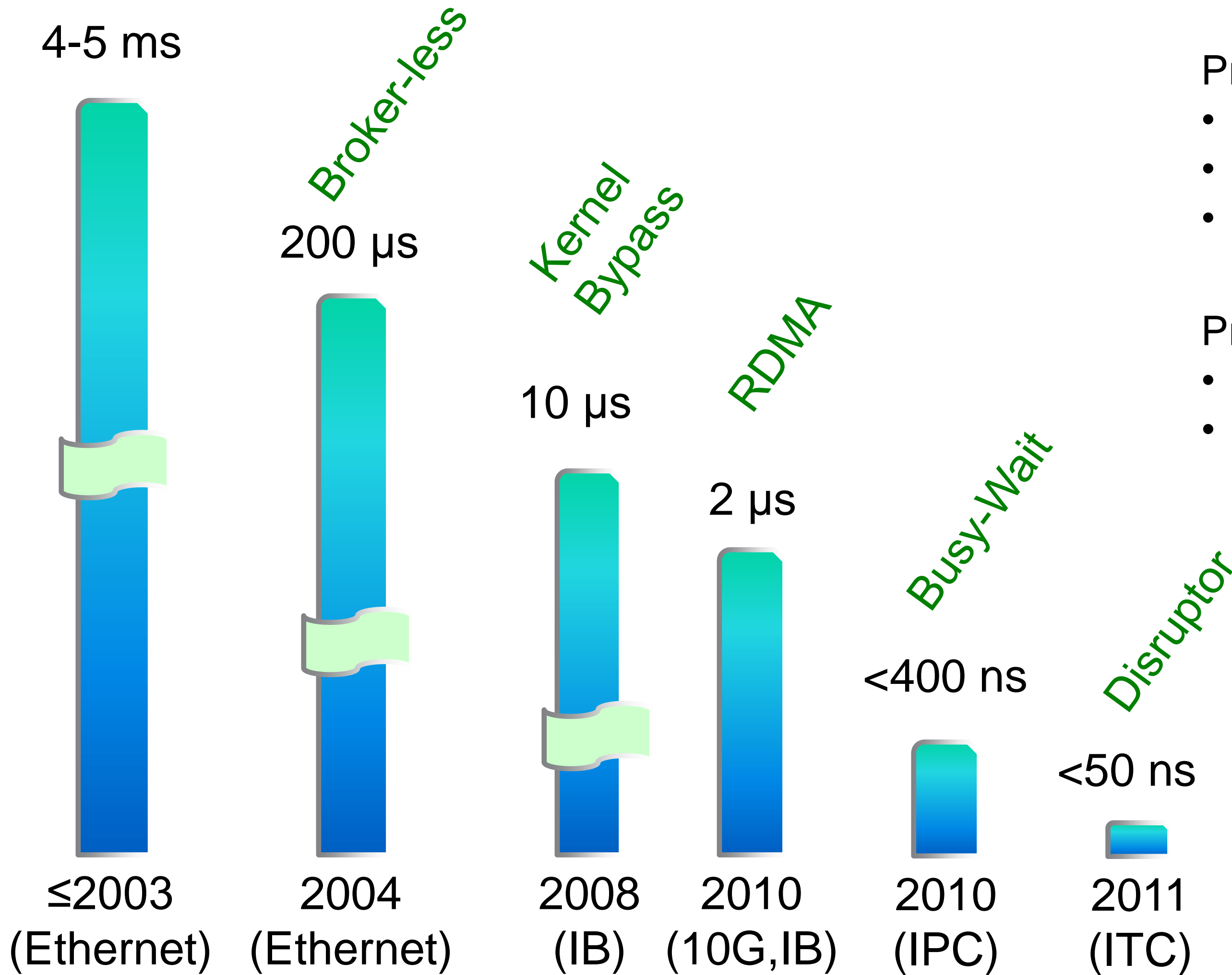
The Exchange

Why Latency Matters



(Ultra) Low Latency Timeline

Race to Zero – Less than 8 years, 10,000x-100,000x decrease!



Predictions – Technology

- <1 µs Eth (2012)
- <500 ns Eth (2015)
- <100 ns Eth (2020)

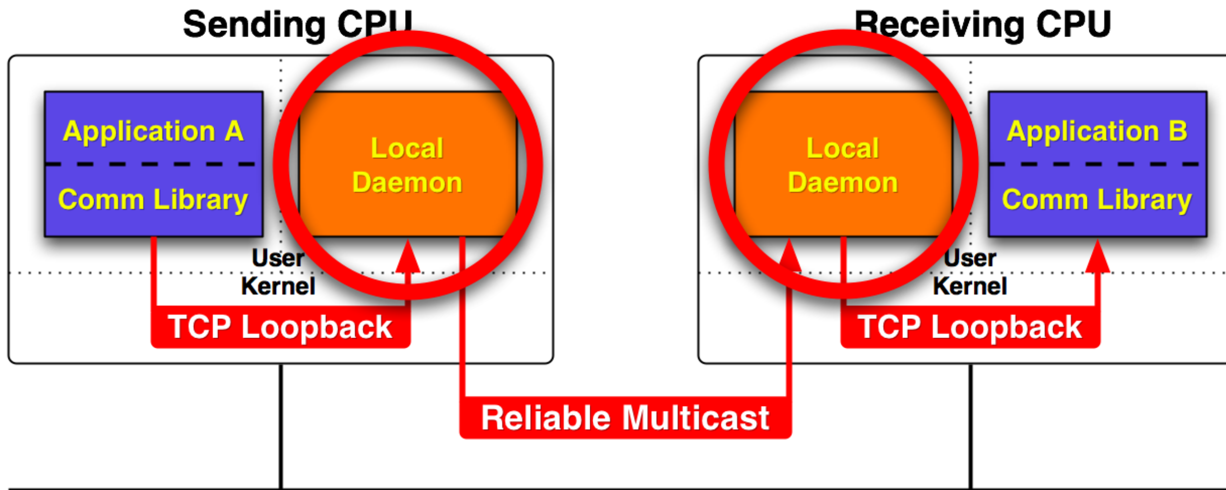
Predictions – Technique

- <100 ns IPC (2012)
- 1G mps ITC (2012)

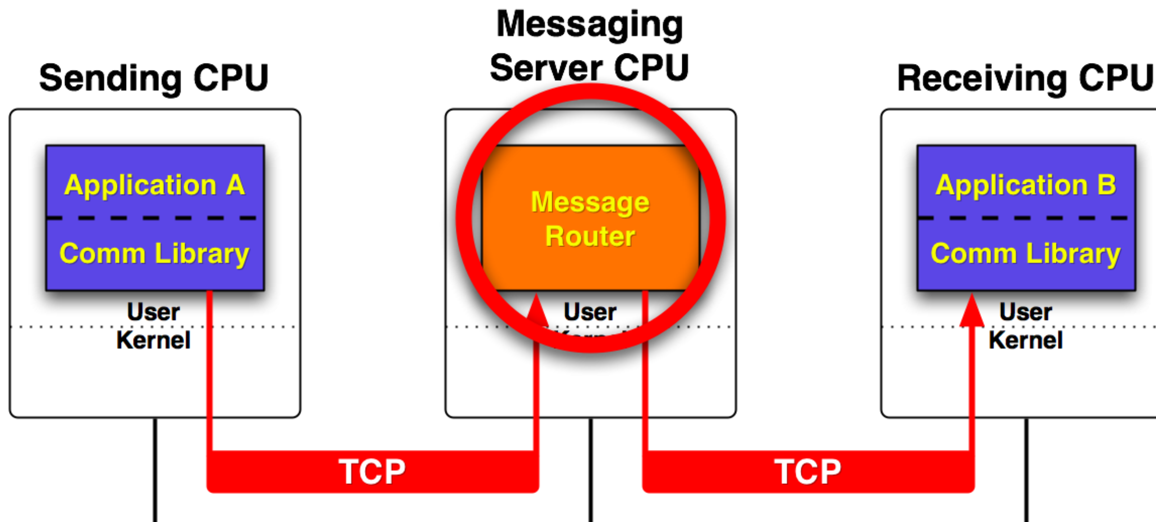
IPC/ITC only
Limited by CPU!

Legacy Messaging Designs

Before 2004



Daemon Based Design
6 Data Hops



Broker Based Design
4 Data Hops

2004 – Need for a State Change

More Efficient, More Scalable, More MORE…

- **Motivations / Challenges**
  - Systems not scaling to todays (yet alone tomorrows!) demands
  - Systems not resilient to failure
  - **Trends:**
    - Need Efficiency, Need Consolidation, More with Less, Need Competitive Advantage (No Vendor Innovation)
- **Broker-based Solutions are a Bottleneck**
  - Broker is a source of contention that limits scaling
  - Broker failure disastrous to latency and stability

Remove the Broker from the Message Path!

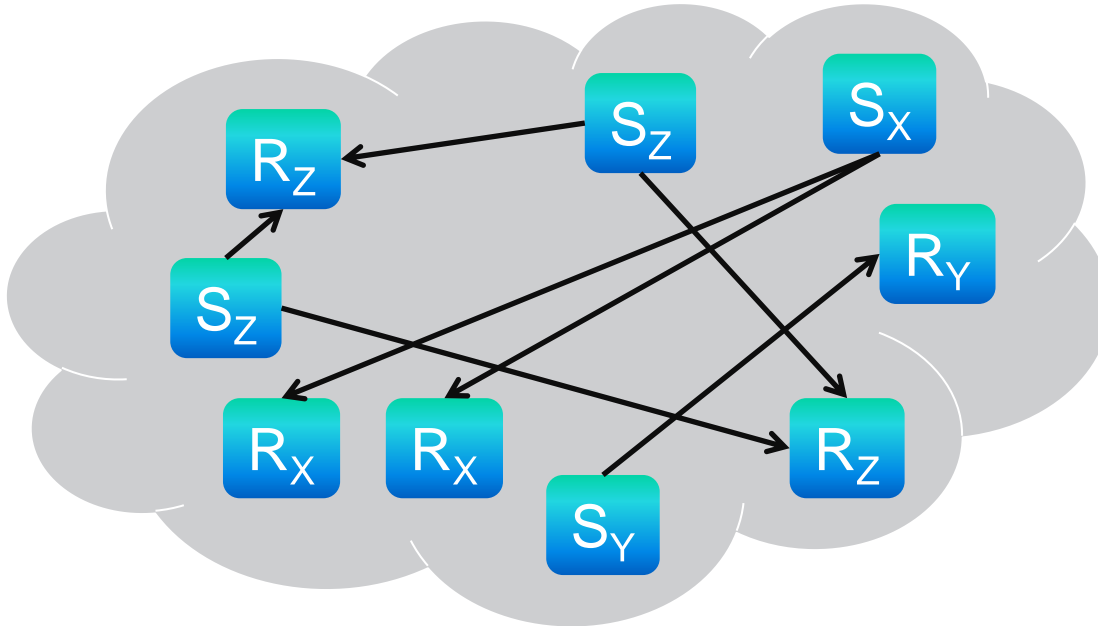
Shared Nothing Messaging

MOM for Todays Demands

- **Peer-to-Peer Messaging**
  - No broker, No daemons
  - Direct connectivity between sources and receivers
- **Parallel Persistence**
  - Broker out of message path and off to the side
  - Broker consulted only for recovery
  - Evolution of Queuing
- **Single Messaging API across all Use Cases**
  - Source-based (vs. Immediate), Event Driven
  - No need for separate Queuing (or PTP) API

Topic Resolution

Connecting Sources and Receivers (Peer-to-Peer)



Traditionally, brokers handled the task of providing transparent connectivity between sources and receivers

Separate the message delivery path and the topic discovery mechanism!

"Service" Location Paradigms

- Static – manual, difficult scaling with topics
- Server-based – (non)caching variants
- Multicast – (un)reliable variants

Avoid including topic string in each message!

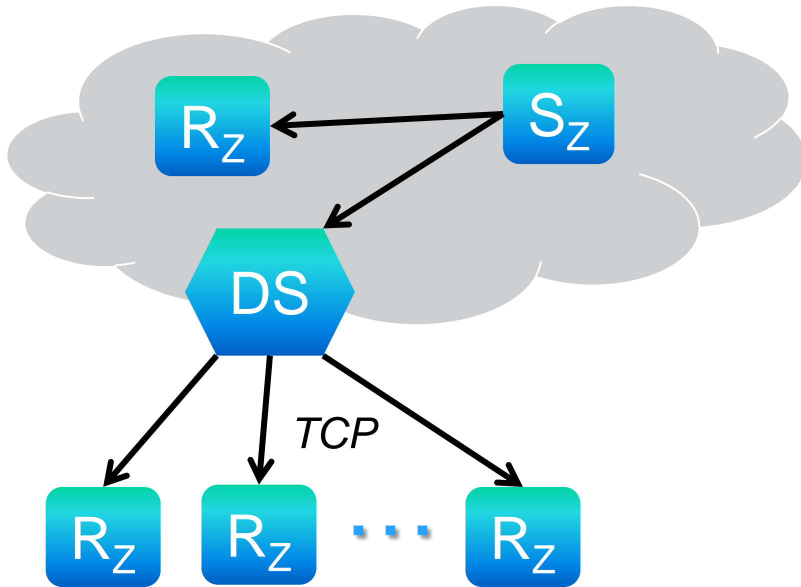
Data Transport Choices

Customization of Connectivity

- **Transport Types – *No One Size Fits All!***
  - **Unicast** (Optimize for single receivers)
    - TCP (with varying buffering behaviors), Reliable Unicast (without congestion control)
  - **Multicast** (Optimize for multiple receivers)
    - (Un)Reliable Multicast (NAK-based)
  - **Intra-Host** (Optimize for lowest latency)
    - IPC (Shared Memory), Inter-Thread (ITC)
- **Source Configuration**
  - Runtime choice

Less Controlled Infrastructures

Architecture for Conflation and Rate Adaptation



Need Per-Receiver backpressure in order to adapt. TCP provides ideal flow and congestion control in these environments and thus ideal backpressure signaling.

All Receivers are Not Equal!

- Desktops
- Web (HTML5/WebSockets Ideally)
- Mobile Apps

Rate Adaptation

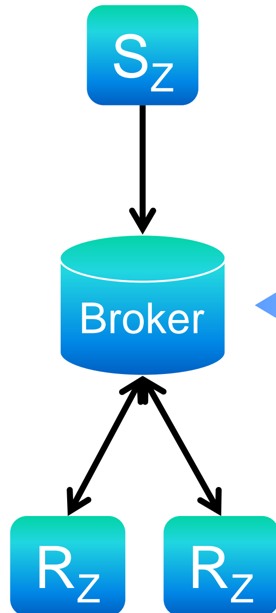
- “Non-”Intelligent Data Drops
  - Tail, Oldest, Head, etc.
- Per-Topic vs. Per-Receiver vs. Per-Connection

Conflation

- Conflate Data from multiple buffered messages into one
- Data Representation Specific

Traditional Persistence

Store and Forward Architecture



Receiver/Delivery Durability

- Receiver can crash or go down gracefully without loss of messages upon restart
- Recovery is the act of restarting and recovering missed messages
- Durability can be extended to Sources also

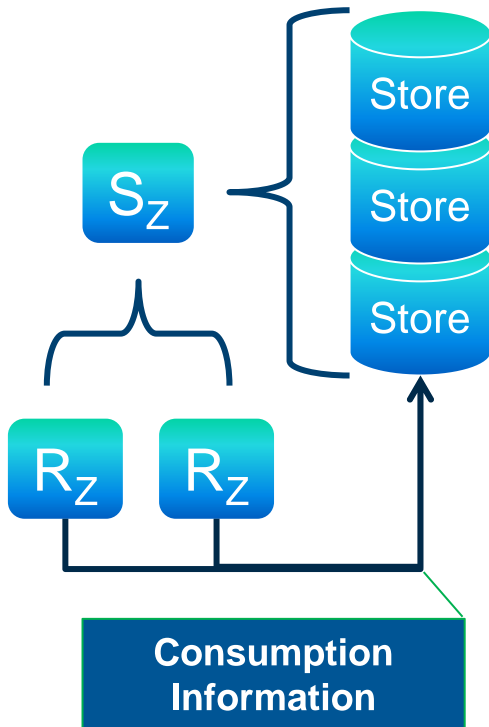
Brokered Architecture Limits

- Broker is point of contention
- Slow receiver impacts source and, more importantly, other receivers
- Broker typically SAN backed (scaling limited)
- Recovery is "pushed" to receiver by broker

Deployments can only scale by adding brokers and splitting the topic space

Parallel Persistence

Durable Delivery without Penalty



Store not in the Message Path

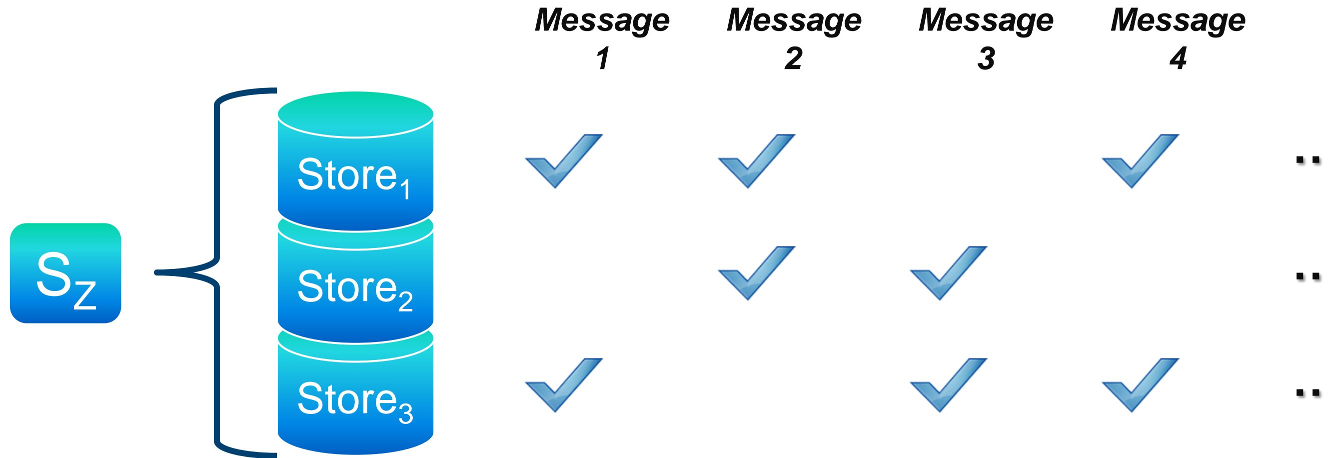
- Stores receive data in parallel to receivers
- Consumption Feedback (ACKs) are out-of-band
- Recovery can occur in parallel to "live" data delivery
- Receiver-driven recovery
  - Receivers pull data from stores
- Stores maintain much less state and do much less
  - No need to track receiver recovery, for example
- Recovery does not impact source or other non-recovering receivers
- Dissemination from source to stores and receivers uses normal peer-to-peer messaging

Store ≠ Broker

Stores do less work, maintain less state, and can scale!

Quorum

Shared Nothing Approach to Persistence



Resiliency

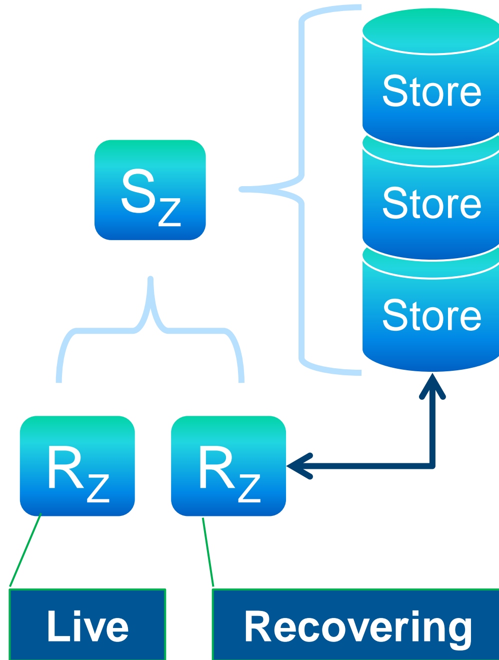
- Avoids “Split-Brain” (majority must be reachable post failure)
- Stores persist locally independently
- Only need Quorum (majority) to withstand failure of minority
- Zero Latency Failover – no need to stop or change behavior

Performance

- Per-Message Striping (+50% per store as shown)

Consensus

Receiver Recovery and Arbitration



Receiver Recovery

- Receivers ask Stores for message consumption status and take majority or highest (arbitration)
- Receivers "pull" messages from stores
  - Load balancing across Stores to spread out impact of recovery
  - Rate of recovery up to individual receivers
  - Rate of recovery not bound by individual store
- Handling the "live" stream from the Source
  - Ignore it or Buffer it (up to individual receiver)
  - Seamless cutover from recovery to live
- Source too fast?
  - Receiver can ignore live stream and pull from stores at slower pace

Messaging API – Sending

Simplifying the Semantics – Publish/Subscribe

Immediate Sends

```
send(“topic A”, data, length);
send(“topic B”, dataB, lengthB);
```

Source-Based Sends

```
srcA = create_src(“topic A”);
srcB = create_src(“topic B”);
…
send(srcA, data, length);
send(srcB, dataB, lengthB);
…
delete_src(srcA);
delete_src(srcB);
```

JMS

Create MessageProducer without Destination and specify Destination on each send

JMS

Create Topic and TopicPublisher

Source-Based APIs

Can leverage Topic Resolution in order to reduce message path latency

Messaging API – Receiving

Simplifying the Semantics – Publish/Subscribe

Event-Driven Reception

How do you handle receiving on thousands to millions of topics?

```
int msg_proc(msg *m, void *cd)
{
  /* handle m based on cd value (rA_state or rB_state)
     and/or m contents */
}
…
rcv1 = create_rcv(“topic A”, msg_proc, rA_state);
Rcv2 = create_rcv(“topic B”, msg_proc, rB_state);
…
```

JMS

Create Topic and TopicSubscriber
Attach MessageListener

Queuing Semantics

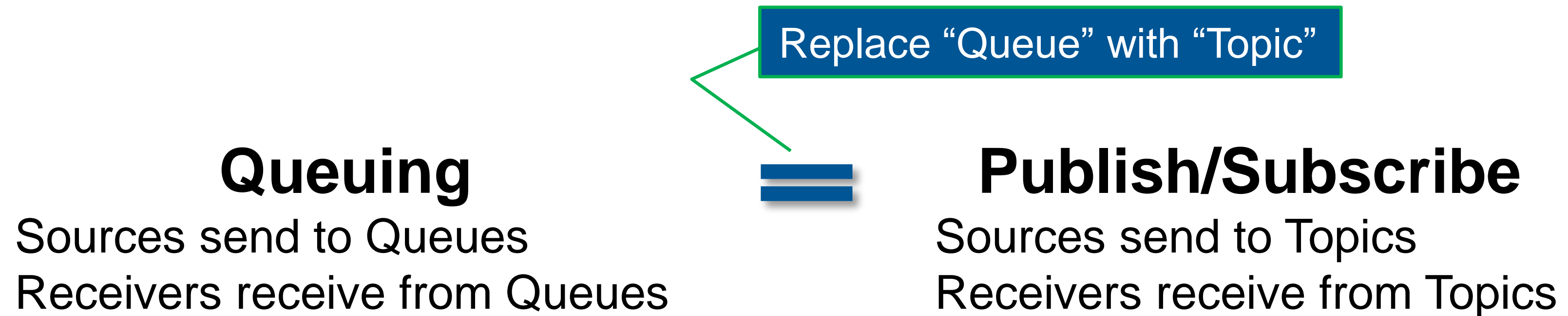
Load Balancing + De-Coupling

- **What semantics are needed for Queuing?**
  - Load Balancing (Once-and-Only-Once)
  - Decoupling
    - Source Rate vs. Receiver Consumption Rate
    - Source Lifetime vs. Receiver Lifetime
- **What APIs are needed for Queuing?**
  - JMS has the Point-to-Point API
    - PTP and Pub/Sub share most calls and interfaces

Does this need to be different than Pub/Sub?!?

Queuing is Dead, Long Live Queuing!

No Need For Point-to-Point to be Different



Single Semantic – Publish/Subscribe

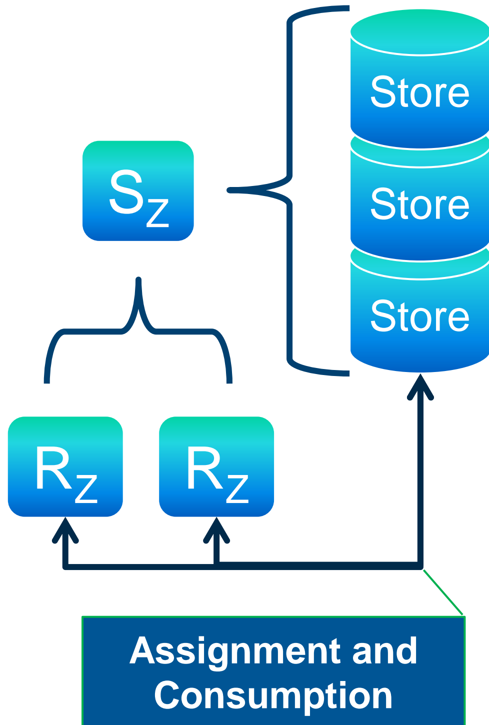
- A queue can be considered a topic
- Need Load Balancing per topic
- Need Rate and Lifetime Decoupling per topic

Point-to-Point API – Redundant

- Subsume the PTP receive call into Pub/Sub

Persistence + Queuing Semantics

Load Balancing + De-Coupling



Load Balancing

- Assignment separate from Data Dissemination
  - Source Assigned
    - Receivers up-to-pace
    - Consumption can backpressure source
  - Store Assigned
    - Receivers request messages (i.e. pull)
  - Assignments sent out-of-band from Data

Rate and Lifetime Decoupling Already Done by Parallel Persistence!

Next-Generation APIs

MOM Evolution

- **Right Value**
  - Not more layers of Abstraction, it’s about Complimentary Functionality
- **Actor Model**
  - Built on message passing
  - Most developers using messaging APIs are already doing it without knowing it
  - Right metaphors for truly useful ESBs

INFORMATICA®
The Data Integration Company™