RTI in Financial Services: Performance, Consistency, Reliability

An RTI Whitepaper

Real-Time Innovations, Inc.
385 Moffett Park Drive
Sunnyvale, CA 94089

www.rti.com

Increases in data volumes over the past few years have stretched current financial information backbones, resetting the technology challenges in a way that demands a new approach.

For example, Automated Trading Desk (atdesk.com), now part of Citigroup Inc., chose RTI to distribute real-time data from direct-exchange and ECN feeds to price-prediction engines and automated trading applications. PIMCO (pimco.com) has selected RTI as part of a new initiative to enforce and monitor regulatory and client-imposed pre-trade investment restrictions.

RTI's solution is a software-only messaging backbone based on a true peer-to-peer architecture that fundamentally challenges earlier generations of daemon-based architectures, and recent peer-to-peer approaches. Leveraging 16+ years of research and development, RTI's key technical value is unparalleled intelligent messaging that performs predictably as message sizes increase and consistently as overall message throughput grows under extreme market conditions. This **deterministic behavior** is unmatched.

With no intermediate daemons, brokers or other components to add latency, market data and market order/execution updates are benchmarking at over **3 million messages per second.** RTI customers have reported mean **latencies as low as 43 microseconds** in Gigabit Ethernet environments. Current tests support **symbol spaces of over 1,000,000 "subjects."** Overall **throughput** is essentially **linear** as publishers are added, with maximum throughput limited only by available bandwidth.

Earlier peer-to-peer approaches for **market data distribution** improve performance and latency over daemon-based architectures, but often experience

large deterioration in performance as message sizes and especially message throughput increases.

For **algorithmic trading and complex event processing**, RTI's key advantage is the efficient mapping of messages to CEP streams. RTI's data-centric infrastructure offers the highest performance interface to the CEP relational model.

RTI delivers secure **enterprise class** performance to bring minimum latency to remote operations globally.

Fundamental Assumptions

Previous assumptions about the middleware infrastructure underlying distributed applications are inadequate regarding the requirements on latency, determinism, throughput, scalability, and availability as the volume of data and the complexity of data flows continue to grow by yet another order of magnitude.

A different approach is needed that sets aside the traditional assumptions. The RTI middleware infrastructure is based on the following assumptions/design principles.

1. **There can be no single point of failure or loading.** Some implementations use a daemon or a broker-based architecture, which can lead to partial failure and complex failure/repair modes during recovery. The RTI infrastructure is based on a decentralized protocol, with no single point of failure, thus minimizing these partial failure situations.

2. **The transport and the network are unreliable.** RTI protocols assume that the media can drop packets, links go down, and hardware fails. This reliability is built into the messaging protocol, which has just recently been formally adopted as an open industry standard[1]. APIs are provided so that applications can be aware of external changes and respond accordingly.

3. **A local cache improves performance and resiliency.** In keeping with the above two assumptions, the RTI infrastructure provides a local *in-memory* cache for all communications.

4. **Topology can change on the fly.** Operational parameters must be changeable on "live" systems without disruption. RTI's messaging and caching infrastructure provides automatic self-discovery and configuration, so that components can be added and removed dynamically on a live system without disruption. For example, servers can be added or removed from a "server pool" without disrupting the clients. Furthermore, there are

[1] Object Management Group (OMG) DDS Interoperability Protocol [http://www.omg.org/technology/documents/dds_spec_catalog.htm]

no startup dependencies imposed by the RTI infrastructure — servers and clients can be started in any order.

5. **Changing requirements demand a highly tunable infrastructure.** RTI's design philosophy has been that a high performance middleware infrastructure should be tunable, so that it can be optimized to the specific application requirements. Thus, RTI exposes detailed configuration and behavior parameters that control every aspect of the messaging and caching infrastructure. The power and flexibility of RTI's solution comes through these "Quality-of-Service" (QoS) controls. Alternative approaches do not expose this level of flexibility and control, making the hidden/implicit assumptions opaque to the developer and non-customizable.

Messaging and Caching Communication Model

RTI's approach is distinguished by the notion of application-defined QoS, the flexible mechanisms for setting up data flows and the lack of centralized resources, daemons or brokers.

The RTI messaging and caching infrastructure is based on a familiar "publish-subscribe" paradigm, with some unique additions and specializations that make it suitable for high-performance real-time client-server and point-to-point interactions as well.

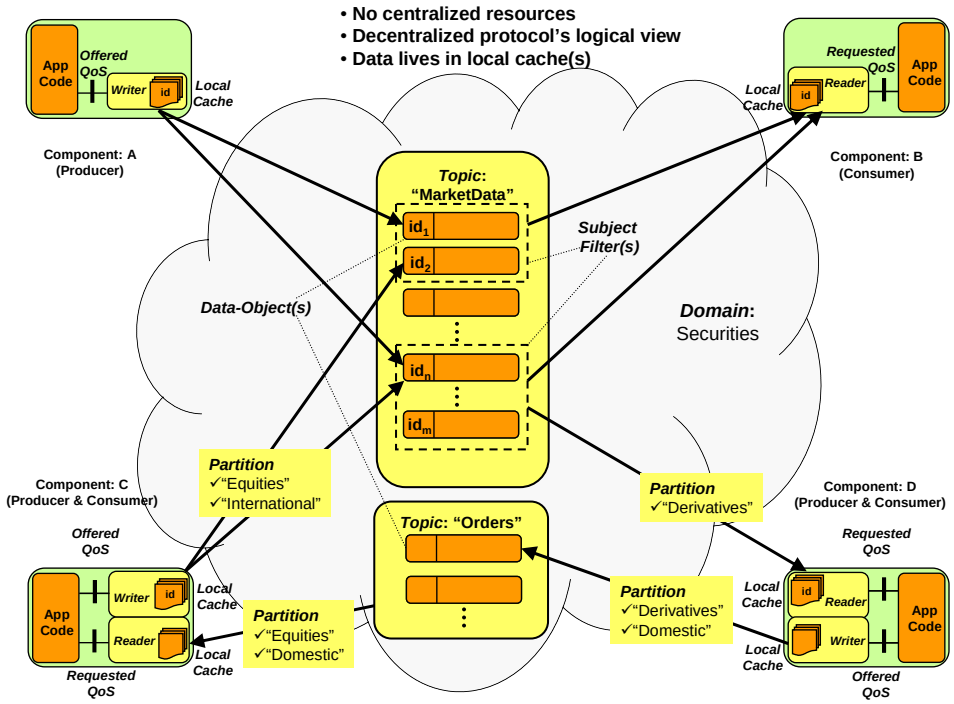

Figure 1. Conceptual overview of RTI's decentralized peer-to-peer communication model. There are no centralized cache or resources, no servers, no daemons. All the data lives in local caches. The underlying decentralized messaging protocol implements a logical "shared whiteboard" view.

The figure provides an overview of the RTI communication model. At the core of the communication model is the notion of a "*data-object*" which refers to some changing real-world entity. For example, a data-object might be stock ticker symbol with changing ask/bid prices (market data). Another example of a data-object might be an order to buy/sell a certain security, with the state changing from created/opened/acked to executed/cancelled.

Producer components (e.g., Component A) use a *Writer* entity to "*publish*" changes to data-objects, while *Consumer* components (e.g., Component B) "*subscribe*" to changes in data-objects via a *Reader* entity. A "message" is a change to data-object, sent from a Writer to Readers. An application component may be both a producer and a consumer for the same or different data-objects (e.g. Components C and D). The RTI infrastructure takes care of delivering the right updates to the right components at the right time, without requiring the components to be even aware of each other.

Data objects are organized into *Topics.* A Topic is a collection of data-objects that all have the same structure and semantics. In Figure 1. C, the "MarketData" Topic is a collection of symbols being traded for which the market data is continuously changing. Similarly, the "Orders" Topic refers to the collection of issued orders.

Within a Topic, a data object is identified by a unique "*id*", which is derived from certain specially marked fields in the data type[2]. Data types are defined for a topic, as well as which fields are to be used as "*keys*" (from which a unique id is derived for a data object).

A Reader or Writer entity is bound to a Topic. Topics are thus a means of associating Reader and Writer entities. Writer(s) communicate changes to the underlying data objects by sending messages to the Readers. A Topic does not "physically" exist in one place. Rather, it is a decentralized notion used to associate Reader and Writer entities, implemented by the underlying decentralized messaging protocol. Readers and Writers have local in-memory caches for buffering the messages describing the changes to the underlying data objects.

A Writer entity can publish updates to one or more specific data objects on the associated Topic. There can be multiple Writers for a given data object. When publishing an update, the Producer always (implicitly or explicitly) specifies the underlying data object being updated.

A Reader entity can subscribe to a subset of the data objects on the associated Topic by specifying a "*Subject Filter.*" A Subject Filter is an expression[3] on the id fields of the Topic's underlying data type. Thus, if a Consumer component is only

[2] Analogous to the *primary keys* in a table in the Database world
[3] RTI supports SQL-92 grammar on the fields of the Topic's data

interested in a subset of stock tickers from the “MarketData” Topic, it can do so by using a Subject Filter for those symbols.

Partitioning

In order for a Topic to associate Readers and Writers, they must all belong to the same “*Domain.*” A Domain is a logical data space that defines the scope of communications; messages in different domains are isolated from one another. A Domain is shown as a cloud in Figure 1. C, and realized by RTI’s decentralized messaging protocol. For example, “Securities” trading might be in a separate domain from “Currency” trading, which in turn might be in a different domain from “Futures” trading. An application component creates a “*Participant*” entity to become a member of a Domain. The Reader, Writer, and Topic entities are locally created from the Participant entity, and belong to it. An application component can participate in multiple domains by creating multiple Participant entities.

Domains are important for effective distribution of real-time information, efficient network bandwidth utilization, and controlled access to different types of financial information.

For partitioning within a Domain, Readers and Writers can be grouped for control of data access via user-defined strings called “*Partitions*”. A partition is an application-defined “tag” on a Writer or Reader entity. Writers and Readers for a Topic do not communicate unless they have a common tag or are untagged. Figure 1. C shows a hypothetical scenario where component C belongs to the “equities” line of business and subscribes to orders that are tagged as (“Equities”, “Domestic”), while it publishes market data updates that are tagged as (“Equities”,“International”). Component D belongs to the “derivatives” line of business and subscribes to market data updates that are tagged as (“Derivatives”), while it publishes orders tagged as (“Derivatives”, “Domestic”). Component C will see updates from component D because they both belong to the “Domestic” partition, however component D will not see updates from component C because they have no common partitions.

Flexible Data Types

RTI provides choices for dealing with data types to suite the needs of a range of application requirements, to shield application developers from the need to know details of data handling and to allow for dynamically changing data types without forcing application components to restart.

Many alternative messaging technologies do not allow types to be dynamically changed without forcing a restart of the application components. Some alternatives do not support strong typing, forcing applications to always manage the marshalling and demarshalling of messages. This creates more work for the

application development team, complicates the application logic and is error-prone.

In comparison, RTI's approach is malleable to an application's requirements. RTI's implementation exchanges type codes as part of the middleware's metadata exchange protocol, allowing applications to discover when there are mismatched type definitions for a Topic. The protocol is extremely efficient and exchanges type definitions only once—when Readers and Writers discover each other—not on a per-message basis.

Ensuring Real-Time Quality-of-Service (QoS)

RTI's infrastructure provides built-in facilities to express and handle requirements such as expressing that the trader wants to ensure that an "automatic trade" order is not open for more than 10 milliseconds.

RTI's messaging and caching infrastructure is designed to deliver not only application data, but also the associated QoS events. Data paths between Writers and Readers on matching Topics are established if-and-only-if the offered QoS is compatible with the requested QoS. If the RTI infrastructure detects that the requested/offered QoS are incompatible, the applications on each side can be notified of this "QoS event". For example, if a reader were to request data faster than can be provided by a writer, then this situation would result in a notification of this "incompatible QoS" event.

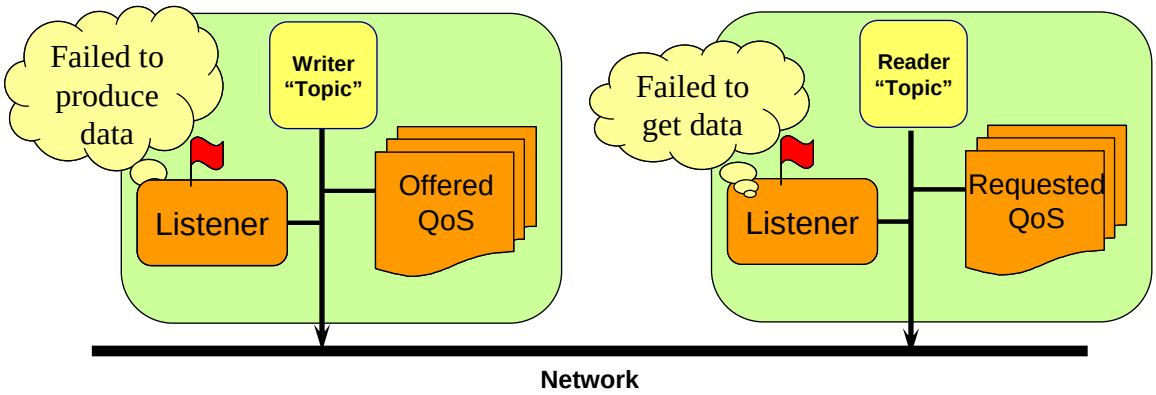

Figure 2. RTI delivers not just application data, but also QoS events.

In addition, when a QoS is not satisfied during operation, the middleware can notify the application of a "QoS event". The figure shows an example where a producer component (left) offers to produce subject updates, say every 1 ms. If the application fails to provide a message update within 1ms, the application can be notified of this QoS event by an associated listener. Likewise, a consumer component (right) may request an update every 10 ms. If an update fails to arrive in 10 ms (say, for the open-order), the application can be notified of this QoS event via a listener (so, perhaps the order can be cancelled). RTI offers a rich set

of QoS policies, many of which come with corresponding listeners that can be used to notify the application when a QoS is not satisfied. The QoS described here is called the *DEADLINE QoSPolicy*, which can be used to achieve real-time operation in a distributed environment.

As a result of this model, the QoS can be **tuned on a per Writer and per Reader** basis. Each Writer-Reader pair establishes independent quality of service (QoS) agreements. This provides fine-grained control so that a given application component can, for example, subscribe to market data, internal analytics, position data, market order confirmations---each with its own specific behavior and recovery patterns. This aspect, unique to RTI, enables application designs that easily support extremely complex, flexible data flow requirements. Ensuring that participants meet the level-of-service contracts enables predictable operation necessary for real-time systems.

Automatic and Dynamic Peer-to-Peer Data Flows

A fundamental design paradigm in RTI is to ensure continued operation in the face of failure. For example, if a link fails and severs a network, each side of network will continue to work independently. When the link is restored, the entire network will recover with minimal system overhead. **Application components can be added and removed dynamically and started in any order.** This is possible because, in RTI's implementation, direct peer-to-peer data paths are automatically established between compatible Writers and Readers. Unlike some other alternatives, RTI's solution does not rely on brokers, servers or daemons. Often, those approaches lead to single points of failure, performance bottlenecks and startup dependencies.

With RTI, the application simply links to an RTI library which implements the de-centralized messaging protocol to establish data flows. Direct peer-to-peer data flows are established in two phases: (1) a discovery phase (also referred to as metadata exchange), in which the application component declarations of the Readers, Writers, QoS are exchanged between Participants over pre-defined **built-in topics**; (2) the delivery phase, in which application messages are communicated. Dynamic changes to Readers, Writers, or their QoS are communicated via the built-in topics in real time.

Data paths are established automatically based on the declarations made to the RTI middleware by application components. Data paths are reconfigured when application components are added, removed or dynamically change their QoS.

In addition, an application can subscribe to the built-in topics to get full **introspection** into the dynamic changes occurring in a Domain. Thus, an Operator Dashboard application can be aware of when a Trader comes on-line or goes off-line and make dynamic adjustments, including bringing other

components on-line to handle the change in system load. Other messaging infrastructures do not expose this level of detail about the system topology.

The **automatic discovery and configuration of data flow** in RTI provides a significant operational advantage for systems with dynamic configuration changes:

- RTI quickly discovers new participants, and automatically establishes the appropriate data flows. The infrastructure cleanly flushes old or failed components and data flows as well.
- Partial failure and startup dependencies are avoided as the application code and the RTI infrastructure libraries are linked together in a single component and run in one address space.

Pluggable Transports

RTI's infrastructure is built on top of a pluggable transport interface. The RTI pluggable transport model does not require the underlying media to be reliable or connection oriented, and is flexible to accommodate a variety of signaling schemes. The reliability protocol is thus built inside the RTI libraries and is configurable via QoS parameters to match the needs of the operating environment.

This pluggable framework approach enables RTI capabilities to be utilized on top of a new transport technology; supports the use of multiple transports simultaneously and enables working through firewalls.

RTI automatically takes care of fragmentation, sequencing, reconstruction and retries of the lost fragments of large messages which exceed the underlying physical transport's "maximum message size" limit. The reliability protocol is designed to understand message fragments and is optimized to resend only the lost fragments. The application programmer is relieved of the burden of dealing with tricky fragmentation issues, especially when working with a mix of different transports technologies.

Custom transports have been developed for switched fabrics, such as StarFabric. The RTI approach can support native InfiniBand switched fabric without incurring the overhead of TCP/IP network stack emulation, as is typically the case with other approaches.

Client-Server and Transactional Interaction

The Client-Server pattern is commonly found in financial services, for example when a Trader's client issues an order execution request to servers and expects a response acknowledging receipt of the order. Additional responses may be issued indicating that the order has been processed and finally indicating the result of the execution. In addition, the position updates resulting from the

processing of the order may need to be distributed not only to the originating Trader's client, but also to many other components, such as P/L management, Risk Management and so on. A load-balanced cluster of servers is often used in this scenario.

RTI's communication model supports the client-server interaction pattern quite naturally and efficiently.

RTI's approach to client-server and transactional interaction patterns leads to:

- Robust Zero-Configuration Deployment
  - No startup dependencies: can start client and servers in any order
  - Robust to link and component failures
  - Redundant servers with no single point of failure
- Higher Performance
  - Maximal Concurrency: can have asynchronous or synchronous calls
  - Maximal Throughput: clients don't waste time waiting
  - Minimal Latency: no polling or connection management
- Scalable architecture

Next Steps

To learn more about RTI's high-performance infrastructure for financial services applications or to request an evaluation, email info@rti.com or visit http://www.rti.com/markets/financial-services.html.