



GemFire Real-Time Events: Technical White Paper

THE RELEVANCE OF EVENT-DRIVEN ARCHITECTURES

While the world we live in has become more event-driven, most technological systems continue to operate in a request/response or a pull-based paradigm. Though this paradigm works fine for client-server applications of the past or even the modern day Service-Oriented Architectures (SOA), it fails to meet the growing need of information consumers, who need access to data as and when a meaningful state change has occurred in the system. In many instances, polling for events has often been adopted as a means to detect such state changes, but this approach has several downsides. First, infrequent polling could result in data latency and performance issues as an application waits too long before it identifies an event. On the other hand, too frequent polling causes severe network congestion especially in environments that have large number clients sending in requests even when no meaningful event has occurred. Finally, the polling code is often rigid, complex and is suited for only certain pre-defined business scenarios. Any time a new scenario has to be modeled, more lines of code have to be added to an already complex framework.

Event-driven architectures provide a convenient mechanism to address these issues. As and when new events occur in the system, relevant clients are notified about any consequent state change. This selective communication ensures network traffic reduction when compared to a standard publish/subscribe mechanism. Moreover, such a push-based communication system implemented using callbacks is more scalable, because nothing is sent until the event is available. Further, flexible event-driven architectures would enable users to dynamically add additional business scenarios that they would like the system to monitor. Message-Oriented Middleware (MOM) can be considered as a baseline implementation of an event-driven architecture for simple events, wherein events that are generated by some application and consumed in its raw form by one or more recipients. The more compelling and challenging use-case pertains to aggregating events from multiple sources (i.e., complex events) in real time along with other related data to make business sense and act. Complex Event Processing (CEP) refers to a science that investigates such use-cases and the techniques needed to address the same. The remainder of the paper will dwell into the details of CEP and how it enables event driven architectures.

CHALLENGES IN STREAM DATA MANAGEMENT AND COMPLEX-EVENT PROCESSING (CEP)

As mentioned in the previous section, there is a new generation of emerging software that can aggregate multiple, detailed level simple events into higher-level "complex" (or derived) events. Such events can be used to identify new opportunities, threats or provide advance warnings to potential problems.

CEP is a core enabler for real-time Business Process Management (BPM) tools and Business Activity Monitoring (BAM) software. Several vendors in this space offer varying degrees of sup-

port for CEP, but none of them can handle the high streaming data volume or low latency requirements as required for applications in industries such as finance, defense intelligence, telecommunications or process control.

There are several challenges in realizing a reliable, high-performance CEP architecture. Some of the crucial ones are listed below:

- 1) **Handle thousands of events per second:** Consider data coming from trading activities of equities (stocks), in particular electronic trading markets such as the NASDAQ. Such markets generate large volumes of data, in bursts that can be in thousands per second. A CEP engine is required to process and analyze events at this high rate. The rate of events at any given time can be highly unpredictable and a good CEP engine should have capabilities to gracefully deal with spikes in message load.
- 2) **Aggregate and correlate data in multiple streams:** A CEP engine should be able to relate and combine data from multiple sources as they arrive, in real-time. This should be possible without introducing any additional latency due to the extra processing involved.
- 3) **Aggregate streaming data with historical or related data:** It is not enough to get a view of just fast moving streaming data. Sometimes, to make sense and enable decision making, applications need views that combine current data with historical streaming data or data originating from other enterprise data repositories.

Architecture

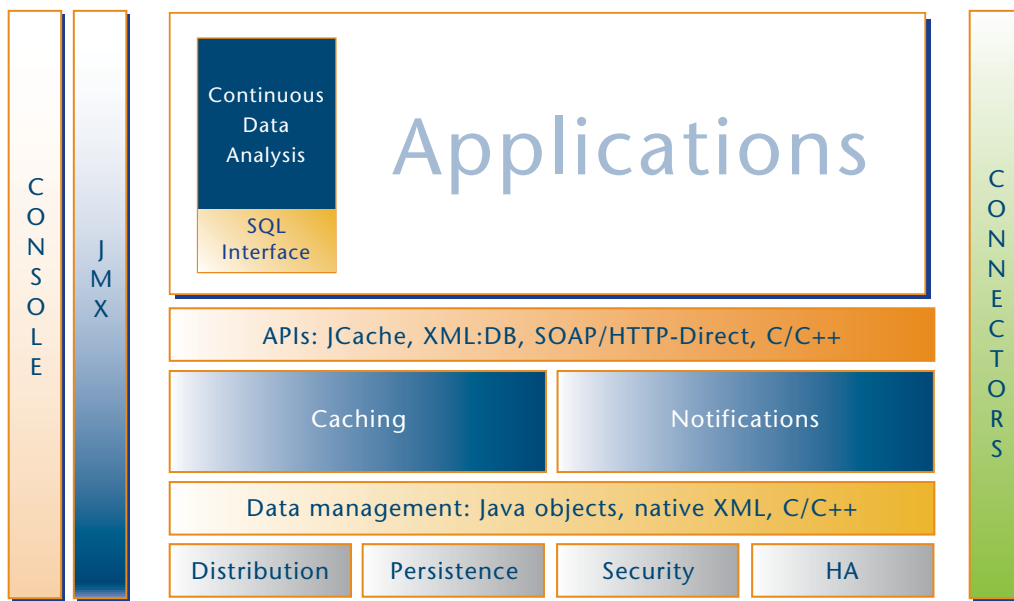


Figure 1: GemFire high-level architecture

4) A language for modeling and analyzing data that is easily understood: Applications need to express their specific interest on the data managed by a CEP engine through a language that is well-understood, easy and offers the flexibility required to express complex conditions, aggregations and correlations. SQL is a language that is de-facto when it comes querying against complex data models. A combination of SQL with extensions to model complex operations can easily serve the purpose.

5) Distributing the derived events to client applications: The CEP engine should not only do continuous analysis of streaming data but should be capable of distributing (pushing) the derived events to remote enterprise applications that may be distributed on a WAN in some cases. Providing resilient client connectivity and re-connection mechanisms are critical functions that a CEP engine must perform.

6) Guaranteed Quality of Service (QoS): For real-time applications that sense and respond to simple events, it is critical that the derived events be delivered within a configured time interval. For instance, in very low latency applications such as in trading, events should be analyzed and results delivered to clients within a few milliseconds.

7) High availability: The CEP engine should provide high-availability for the data it manages through flexible replication strategies. A highly available engine should also ensure that all run-time components are resilient and provide automatic fail-over services.

GEMFIRE REAL-TIME EVENTS: AN OVERVIEW

Based on the concepts presented earlier, one can easily infer that ensuring speed and agility in business activities directly depends on an organization's ability to sense and respond in real-time to critical events. In fiercely competitive markets like Financial Services, which are characterized by high volatility and reducing margins, sensing and reacting to events in that ecosystem often provides the ability to not only capitalize on opportunities that would have otherwise gone unnoticed, but also identify a company's vulnerabilities and execute accordingly. Several other industries like federal intelligence, telecom and manufacturing/logistics have similar applications for such event-driven business models. GemFire Real-Time Events (RTE), a core component of the GemFire Enterprise Data Fabric (EDF) is a CEP engine that enables an enterprise to facilitate agile, event-driven business models. It helps companies:

Instantly identify business events that are relevant to them based on real-time information that is constantly changing.

Immediately analyze these events to discern patterns and scenarios of interest, with the ability to correlate with other sources of information like historical data, customer data, reference data, etc.

Intelligently distribute appropriate information to relevant clients and applications that have to react to these business events.

From a technology standpoint, RTE is a main-memory based active data management system that can analyze thousands of events per second and distribute events of interest to hundreds of remote clients. The system consists of one or more server processes that hold and process data for any number of remote client processes. The processes create and populate the RTE server, and clients run queries against the data. The clients can perform two types of queries: ad-hoc SQL queries, and GemFire Continuous Queries (CQ) that are registered with the server and automatically updated as the data changes. Traditional query processors utilize a request-response paradigm whereby a user poses a logical query against data and a query engine processes that query to generate a finite result set. If this finite result set is cached then it becomes stale quickly, especially when the underlying data is continuously changing, as in the case of streaming data sources.

In the continuous query paradigm, the users register logical specifications of interest over streaming data sources, and a continuous query engine filters and synthesizes the data sources that deliver streaming, unbounded results to users. RTE is a continuous querying engine implementation that acquires the incoming streaming data into one or more server-side caches and continuously analyzes the updates to the tables to determine how the different queries registered are affected. Subscribing clients in GemFire use SQL statements to express their data set of interest. All the delta events are automatically pushed or pulled to/from the client caches periodically keeping their state synchronized at all times.

SYSTEM ARCHITECTURE

The system architecture of GemFire Real-Time Events is depicted in Figure 1. It comprises of the RTE server, which includes an embedded in-memory database and a number of client applications working to configure, populate, and query application data within the server. As mentioned before, Clients can execute SQL queries and they can also register continuous queries with the server. These queries can be configured with listeners to receive updates to the query result sets at regular intervals. The events supplied to the listeners include row delta information that is supplied to listener callbacks and is used to keep the ResultSets up-to-date. The client can also explicitly request a result set at any time for any CQ that it has registered with the server. The server and client processes can be distributed over any number of machines.

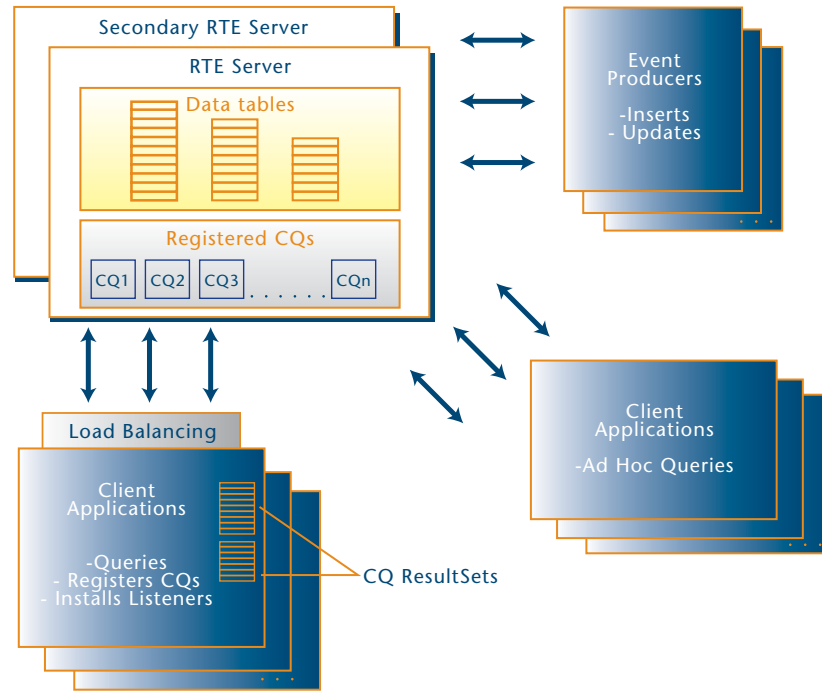


Figure 1: System Architecture of GemFire Real-Time Events

To a great extent, RTE's architecture has been anchored on the need to overcome the challenges in the realm of Complex Event Processing (CEP) outlined previously. RTE's sophisticated in-memory architecture enables it to process several thousands of events per second and analyze them in the context of queries registered by the users. Further, several event streams can be processed instantaneously with no latency as RTE can act as a receiver for multiple event producers. Further real-time information can be correlated with other sources of data (like historical information) to enable evaluation of complex business queries. As can be seen in Figure 1, client applications can register queries (CQ or ad-hoc queries) through standard SQL, without the need for proprietary semantics. Queries impacted by a new set of events are identified and updates and changes are published to the relevant clients (i.e., those impacted by the new events). Based on QoS requirements, clients can determine the frequency at which these updates are published to them. Multiple RTE servers can be instantiated to load-balance client requests and also to ensure high availability upon failure of the primary server.

TECHNICAL HIGHLIGHTS

Intelligence for handling hundreds of Continuous Queries: RTE uses several algorithms and techniques for managing query predicates to quickly determine the queries that are affected by incoming events and also supports advanced joins and incremental client-view maintenance.

Data distribution and notification services: All events captured as deltas on the RTE server are translated in real-time into deltas on views registered by clients and pushed to them. Data can be distributed to hundreds of clients in real-time. Clients are notified of in-coming data through simple call-backs.

High availability: The RTE server can be configured to be highly available. GemFire accomplishes this by replicating data as it arrives at one or more servers. All meta-data that the server uses, such as information on active queries and schema information, is available to all servers. Clients are automatically routed to alternate servers if the server to which they are connected fails.

Simple Intuitive syntax and semantics: RTE provides a very intuitive and easy to use interface for application developers. All events are captured using simple SQL Insert, Update and Delete operations. Applications express interest in data through views expressed also using SQL select statements.

Load balancing: RTE can be configured to load balance connections across several replicated servers. Load balancing will result in the servers performing and scaling better. Clients will have the choice of using sticky or round-robin load balancing schemes.

Automatic fail-over: When configured with multiple replicated servers, RTE will automatically detect unrecoverable errors in server connections and automatically delegate to alternate servers.

BUSINESS APPLICATIONS

Processing of streaming events is a field that has relevance in several different industries. It is no doubt a technology that is still in its early days, but it does offer significant potential for growth, profitability and risk reduction in multiple business environments. These include -

Capital Markets: Financial markets are often notorious for the deluge of data that gets generated every second. Further, financial institutions are often burdened with the challenge of managing data streams from multiple trading venues and understand their business impact. A technology like RTE can help such firms monitor multiple market data streams for patterns of interest (for e.g., "Notify when IBM > 100, and S&P moves by 2%") and execute profitable trading strategies based on these patterns. Further, real-time analysis of streaming data also enables

firms to accurately sense the impact of market movements on their exposure and positions to manage risk appropriately. Through real-time monitoring of trading data, compliance violations as well as fraudulent activities can be detected instantaneously.

Federal Intelligence: Intelligence operations in battlefield or even in general security enforcement can significantly benefit from the ability to analyze real-time information feeds for specific scenarios. Usage of geo-spatial analysis in such scenarios could greatly assist in tracking events like platoon movements and enable quick decision making at control centers. Real-time processing of data streams becomes relevant in electronic signal intelligence applications as well as in analyzing message traffic for special patterns. Further, any such analysis should be immediately propagated to different clients spread across a WAN, where in a solution like RTE can add tremendous value.

Fraud Detection (Credit Card Processing and Wireless Networks): With the growing popularity of wireless services and credit cards, there are consequent problems relating to fraud, identity theft, etc. that need to be addressed. Solutions like RTE can be used for real-time analysis of the a maze of financial transactions or network events like Call Detail Records (CDR) and IP Detail Records (IPDR) as the case may be, to notify relevant systems about fraudulent activity. In addition to analyzing the thousands of events, the event processing system has to correlate these records with historical data and subscriber profiles instantaneously to obtain a real-time view of customer activity. The high-performance analysis and correlation capabilities of RTE will be of potential value in these environments.

RFID (Manufacturing/Distribution/Logistics): The heart of an RFID ecosystem is an EPC (Electronic Product Code) network that comprises of RFID tags, readers and middle-ware systems also know as EPC edge servers. Readers can detect tags and emit relevant data, which is then analyzed by an 'edge' server and suitably propagated to back-end systems (ERP, Inventory, production systems, etc). The large volumes of data emitted by the readers causes the need to instantly process this information and send only relevant information to the other systems of the supply chain, thereby avoiding a data bottleneck. Hence, an RFID middle-ware solution can greatly benefit from a solution like RTE that can analyze, filter and selectively route data emitted from the readers. It is important to perform this analysis as close to the readers as possible so that unnecessary data is filtered out of the supply chain as soon as possible.

BUSINESS APPLICATIONS

The following example illustrates a client programming model for RTE using the JDBC API.

Clients, event producers or consumers, register with a GemFire RTE system by acquiring a JDBC connection to any of the servers participating in the distributed RTE system. The JDBC driver automatically establishes connections to all other members of the system for load-balancing and failover purposes.

```

// Register the JDBC Driver.
Class.forName( "com.gemstone.gemfire.sql.jdbcDriver" );
// Get a JDBC Connection.
Properties endpoints = new Properties();
String host = System.getProperty( "host", "localhost" );
String port = System.getProperty( "port", "30303" );

endpoints.setProperty( "endpoints", "endone=" + host + ":" + port );
// an Endpoint refers to one or more RTE servers that can be used to load balance
requests.
Connection conn = DriverManager.getConnection( "jdbc:gfsq:", endpoints );

```

Event consumers register CQs and listeners to receive notifications on result set modifications.

```

// Get a CQManager for handling continous queries...
CQManager manager = CQManager.getCQManager( conn );
// Register a query for updates every 500 milliseconds...
CQ cq = manager.create
    ("SELECT trade_id, symbol, qty FROM Trade WHERE symbol = 'IBM' AND qty
    > 20" );

cq.setUpdateInterval( 500 );

```

Applications configure the refresh interval based on the ability to handle events at a certain rate. A trader desktop application that renders the results on a chart doesn't need to refresh less than a few hundred milliseconds.

```

cq.setCQListener( new MyListener() );
ResultSet resultSet = manager.register( "symbols", cq );

```

The first time the CQ is registered the result set reflects the current state in the server. All future modifications to the result set are automatically pushed to the client node and merged into the cached result set.

Producers of data events simply dispatch SQL DML ("data manipulation language") statements to the server.

```
Statement st = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,  
ResultSet.CONCUR_READ_ONLY );  
.....  
String sql =  
    "INSERT INTO TRADE(trade_id, qty, symbol)  
    VALUES("+id+", "+qty+", "+sym+"");  
int status = st.executeUpdate( sql );
```

Callbacks are received on the registered listener once every 500 ms, guaranteeing a 'response time' QoS (Quality of Service).

```
public static class MyListener implements CQListener {  
  
    public void beforeResultsUpdated( CQUpdate theUpdate ) {  
  
        < Application logic >  
  
    }  
}
```

CQUpdate encapsulates deltas to result set that have occurred since last "execution" of the CQ. A set of RowDeltas indicate how the result set has changed.

```
public void afterResultsUpdated( CQUpdate theUpdate ) {  
  
    RowDelta[] deltas = theUpdate.getDeltas();  
  
    for( int i = 0; i < deltas.length; i++ ) {  
        RowDelta rowDelta = deltas[i];  
  
        switch(rowDelta.getType()) {  
            case CQUpdate.INSERT: {  
  
                Object[] newData = rowDelta.getNewRow();  
                < Application logic >  
                break;  
            }  
        }  
    }  
}
```

```

case CQUpdate.UPDATE: {
Object[] newData = rowDelta.getNewRow();
Object[] oldData = rowDelta.getOldRow();
    < Application logic >
break;
}
case CQUpdate.DELETE: {
Object[] delData = rowDelta.getOldRow();
< Application logic >
break;
}
}

public void queryReregistered(ResultSet rs) {
    < Application logic >
}
    
```

As the RTE system captures historical data as well, applications can execute ad-hoc SQL queries on the server as well.

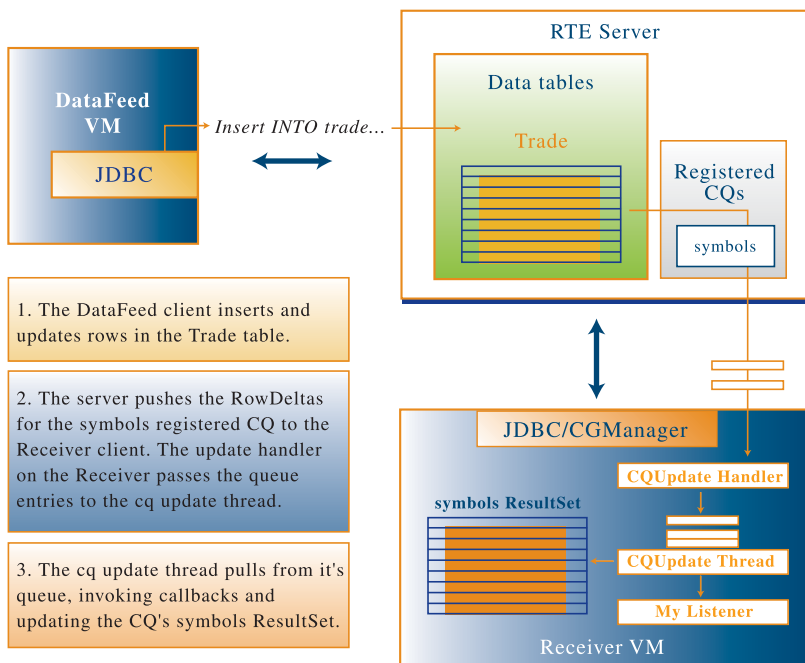


Figure 2: RTE Application Model

PERFORMANCE OPTIMIZATION HIGHLIGHTS

Event Conflation

As streaming events can arrive at an unpredictable rate, a sudden increase can result in a deluge of derived events being propagated to client application nodes. The receiving application nodes may not be able to process events at the same rate as the sender - either due to lack of CPU processing power or due to the nature of the application design. A burst of derived events routed through internal enterprise networks could also easily saturate the network. The chain of events could ultimately result in events getting backed up in the servers. To accommodate such circumstances, client applications can fine-tune the event delivery rate by specifying a refresh interval during CQ registration.

The server intelligently uses a combination of the refresh interval across all registered CQs and arrival rate from each input stream to figure out the rate at which it processes events on the server. Events coming in a rate much higher than required by the clients are automatically conflated - i.e. some of the update events are simply shed. This load-shedding technique allows the servers to scale well and maintain a predictable quality of service.

Multi-Query Optimization

The RTE server handles thousands of events every second with hundreds of continuous queries registered. The engine has to rapidly analyze the event against many query filter conditions to keep up with the volume of data. Among many techniques, the RTE engine uses a technique referred to as Multi-query optimization where it exploits the dependencies among all registered queries to derive a small set of query evaluation plans. For instance, many queries may involve the same joins. The optimizer does the join once and intermediate results for several queries can be formulated in one shot. RTE server organizes complex queries conditions as many simple query conditions, and with the advance knowledge of all interdependencies across queries, it is able to deduce the impact to many queries through just a few condition checks.

"Delta" calculation and propagation

RTE clients maintain the result set for each continuous query that is refreshed in real-time by servers. As events flow through the RTE server, the engine not only computes the queries that have been impacted but also the exact nature of the change to the cached result set in the client. The engine calculates this "delta" in real-time for each client's result set and distributes it to the respective client nodes. This fine-grained distribution minimizes network traffic without compromising on the simplicity of the programming model.

Dynamic revision of query results

Most stream data management systems assume an append-only model in which the derived events from the stream processor are notified to the client application. It is left to the applica-

tion to use the event it whatever way it desires. But if an event gets dropped or contains incorrect data, applications are forced to live with approximate or imperfect results. RTE manages views of queries on the behalf of the client application and is capable of merging any revisions to the results automatically, preserving the correct view of a query at all times.

Materialized view maintenance

Data joins are an expensive operation for a stream data processing engine, especially on large data sets. To mitigate the effect, materialized views can be used to pre-join the data at arrival time, avoiding the join cost at query evaluation time. This approach is increasingly being supported by commercial databases and data warehouse systems. RTE supports a similar concept where the developer can specify continuous queries as views to be materialized at registration time. As underlying tables changes occur, the view is incrementally maintained according to the specified join conditions.

Rather than maintaining multiple copies of data (in the base table and the materialized view) in memory, the RTE server uses efficient reference management techniques to minimize the memory overhead.

HIGH- AVAILABILITY (HA)

As RTE manages data in-memory, ensuring that data is protected against virtual machine or hardware failures becomes paramount. RTE guarantees high-availability of data and the server machinery, through a combination of replication, client side replay of server messages, transparent fail-over and dynamic propagation of membership views to all clients. This section provides an overview of the RTE system provides HA and fail-over.

A highly available RTE system consists of two or more replicated servers. The system automatically assigns the first server (member) as the primary and all subsequent servers as secondary servers. All data updates are always routed to the primary server, which then asynchronously replicates the updates to all secondary servers. The asynchronous replication allows the primary server to scale well, minimizes network bandwidth through batch distribution and avoids bottlenecks in case any of the secondary servers are not equally responsive.

When a primary server fails in RTE, the remaining servers work to ensure data consistency. Each client maintains a rolling cache of update events that have been executed on the primary server, and every attempt is made to ensure that the new primary server has all the events before it becomes available to the system as a primary. This may require clients to replay past messages to the new server so that it can apply events without the loss of data. Replay handling is done so that messages are replayed to the new server in the same order that the old primary applied the events before crashing. The replication system is designed to make sure that data or consistency loss never occurs.

When any server fails the CQs registered with the server are automatically re-registered with other servers. A recovered server is restarted just as any other server is. When the server comes up, it obtains a snapshot of the system from the primary server. Only then does the recovered server announce itself to clients, which can then register queries on that server. Following the general rules for server startup, the server is placed last in the list of secondary servers.

The complete fail-over mechanism is handled transparent to the client applications.

SCALABILITY

Many system variables affect overall throughput and latency performance of the RTE system. For example, the total number of CQs registered has an impact on overall system throughput. How much memory the servers have available for data management and query processing also affects system response times. In general, system performance is impacted by variables like the number of rows in the various tables involved in CQs, number of registered CQs, join processing costs, result set size, width of tables, number of remote clients and the data arrival rate. Thus, for applications with high demands on the server, RTE incorporates a policy-driven load-balancing scheme. RTE supports the following balancing policies:

- ***Sticky***

In this mode, the client establishes a connection to a single server and all traffic from that particular client is sent on that connection. If requests time out or produce exceptions, the client JDBC driver picks another server and then sends further requests to that server. This achieves a level of load balancing by redirecting requests away from servers that produce timeouts.

- ***RoundRobin***

In this mode, the client establishes connections to all the servers in the server list and then randomly picks a server for each given request. For the next request, it picks the next server in the list.

- ***Random:***

In this mode, the edge establishes connections to all the servers in the server list and then randomly picks a server for every request

ADMINISTRATION AND MONITORING

RTE includes different methods to administer, debug, monitor and analyze a RTE system. These are:

CONSOLE: This is a web browser based tool to administer, monitor, inspect and analyze a deployed RTE system and its resources - individual servers, CQs and data.

LOGGING: Logging can be turned ON for each server individually and provides a way to trace the system operations. RTE enables logging to be turned ON at various levels ranging from just configuration information to very detailed logging.

STATISTICS: In addition to logging, RTE gathers a lot of statistical information such as number of active clients, number of CQs registered per client, average time for processing events, average time for CQ processing, distribution message statistics, etc, providing a very fine level of monitoring. The various system statistics are captured in memory and a daemon thread sweeps the stats at configurable intervals and archives these to a "statistics" file on local disk.

RTE provides a graphical tool to chart in real-time the various statistics on a time axis, correlate these with one another to pin-point hard to spot problems very easily and quickly

The screenshot displays the GemFire Real-Time Events Management Console. The top left features the GemFire logo and the text "Real-Time Events Management Console". The top right shows a welcome message: "Welcome gfrtoadmin! Click on a resource from the tree view to explore the GemFire Real-Time Events system." The main interface is divided into a left sidebar and a central content area. The sidebar shows a tree view under "System" with "GFRTE Servers" expanded to show "jackf-desk:30303 (Primary)" and "CQs". The central content area has a "System" header and two tables. The "Configurations" table lists: Multicast IP (24.0.0.250), Multicast Port (7011), Primary GFRTE Server (jackf-desk:30303), and Start time (Tue Feb 15 17:10:02 PST 2005). The "Resources" table lists: GFRTE Servers (1) and CQs (20). At the bottom of the central area, there is a "Refresh" button and "Last Update Time: 17:47:10".

Figure 3: GemFire RTE Console

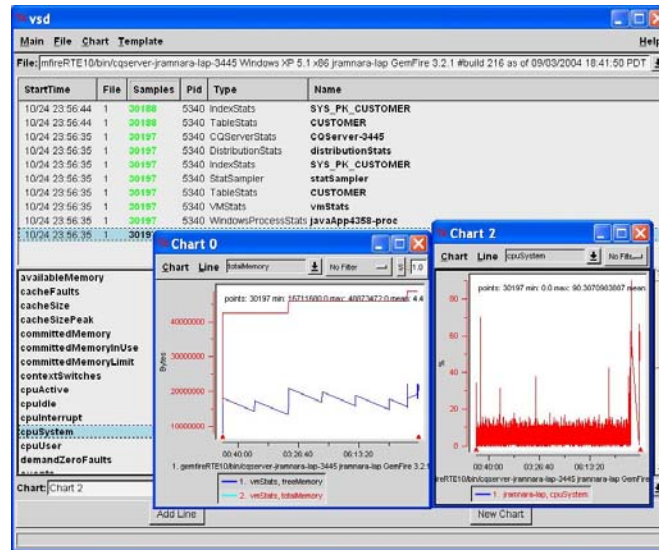


Figure 4: Visual Statistics Display for RTE

SUMMARY

As the importance of business event processing continues to grow, adoption of technologies like RTE would become of strategic importance for a dynamic IT organization. It is important to consider the challenges and business relevance of such initiatives carefully. While the business examples stated in this document are illustrative applications of such a technology, there are certainly several more use-cases that one could identify based on careful introspection of any business and IT ecosystem. GemStone Systems with its GemFire Real-Time Events can help you in such a discovery process and ease your adoption of event-driven architectures.

Corporate Headquarters:

1260 NW Waterhouse Ave., Suite 200 Beaverton, OR 97006 | Phone: 503.533.3000 | Fax: 503.629.8556 | info@gemstone.com | www.gemstone.com

Regional Sales Offices:

New York | 90 Park Avenue 17th Floor New York, NY 10016 | Phone: 212.786.7328

Washington D.C. | 3 Bethesda Metro Center Suite 778 Bethesda, MD 20814 | Phone: 301.664.8494

Santa Clara | 2880 Lakeside Drive Suite 331 Santa Clara, CA 95054 | Phone: 408.496.0242

Copyright© 2005 by GemStone Systems, Inc. All rights reserved. GemStone®, GemFire™, and the GemStone logo are trademarks or registered trademarks of GemStone Systems, Inc. Information in this document is subject to change without notice.

