Case Study

Real-Time Ad Bidding

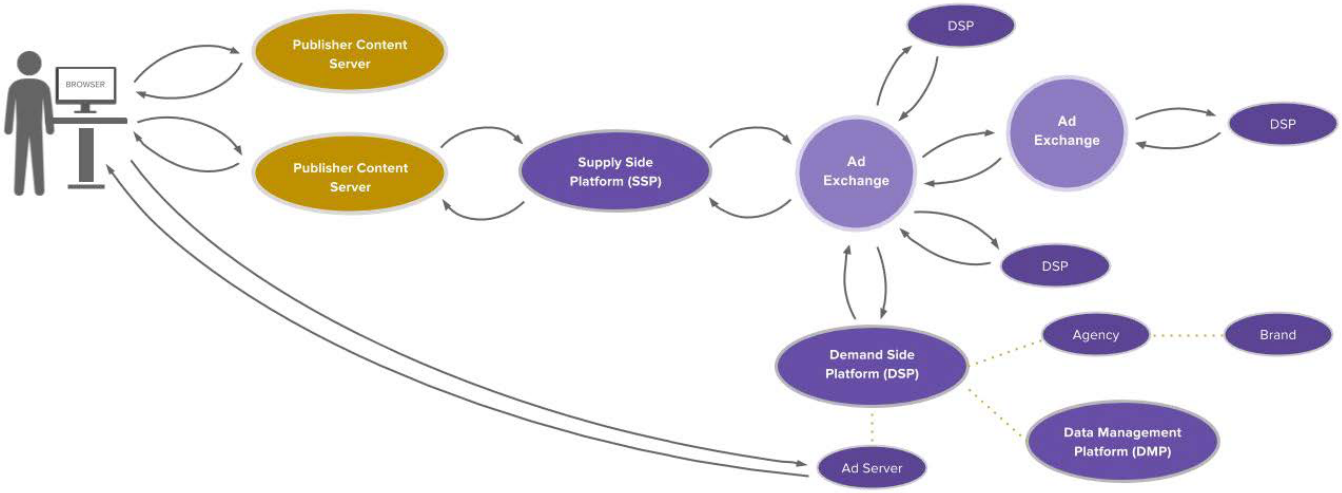
EXECUTIVE SUMMARY

This case study shows how N5 Technologies used Rumi™, its high-performance application platform, to build a real-time ad bidding (RTB) system. By replacing traditional, complex stacks with Rumi's message-driven and hyperconverged design, the team achieved high performance, linear scalability, fault tolerance, and simpler operations—at significantly lower cost and footprint. It outlines the limitations of older architectures and highlights Rumi's impact through benchmark results.

A BACKGROUND OF RTB SYSTEMS

Real-Time Bidding (RTB) is the process of buying and selling online ad impressions in real time through automated auctions. Much like financial markets, each ad opportunity is auctioned off as a page loads, allowing advertisers to bid on individual users based on demographics, interests, and behavior—a practice known as addressable advertising.

Here's how a typical RTB flow works when a user opens a webpage with an ad placeholder:



1. **Initial Ad Request**
   The user's browser loads a URL tied to the publisher's ad server. If the ad space is reserved by a direct buyer, the ad is served immediately.

2. **Unreserved Inventory**
   If the space isn't reserved, the publisher's ad server contacts a Supply-Side Platform (SSP) to offer the space. The SSP may enrich the request with user behavior or page context.

3. **Ad Exchange & Auction**

The SSP forwards the request to an ad exchange. The exchange may consult a Data Management Platform (DMP) for visitor insights, then broadcasts bid requests to Demand-Side Platforms (DSPs) or other buyers. Some DSPs may have pre-set conditions for automatic bidding—similar to limit orders in stock trading.

4. **Bid Response & Ad Serving**
   Bidders must respond quickly—often within tens of milliseconds. The ad exchange selects a winner, debits the bidder's account, and sends a win notification. It also returns the ad URL (either included in the bid or sent later) to the SSP.

5. **Ad Delivery**
   The SSP passes the URL to the publisher's ad server, which tells the browser how to retrieve and display the ad from the advertiser's server.

In some cases - such as small publishers using platforms like Google AdSense - the SSP interaction happens directly from the browser, bypassing the publisher's ad server.

NON-FUNCTIONAL DEMANDS OF RTB SYSTEMS

By its very nature, ad bidding is a high-stakes, time-sensitive process. To be viable, these systems must meet extremely demanding non-functional requirements:

- **Performance**: Bids must be resolved in milliseconds—or faster—to avoid delays noticeable to users. Since multiple ad slots may be auctioned simultaneously as a page loads, the system must sustain high throughput and concurrency while keeping latency imperceptibly low.

- **Reliability**: Every bid must be processed without loss, even in the face of hardware, software, or network failures. This reliability must span all components involved in the end-to-end bidding flow to ensure both consistency and correctness.

- **Scalability**: Traffic to ad platforms is inherently bursty. To cope with sudden spikes—often driven by time of day, geography, or campaign activity—systems must scale horizontally and elastically without degradation in performance.

Meeting these service-level expectations is no small feat. Doing so with traditional architectures often requires intricate, low-level programming and infrastructure coordination, which increases risk and slows down the delivery of business logic. As a result, many teams struggle to balance agility with the stringent performance and resilience demands of modern RTB platforms.

CHALLENGES WITH PRIOR RTB IMPLEMENTATIONS

Complexity

Prior systems were inherently complex to build, test, and operate. Developers often had to construct nearly every aspect of the infrastructure from scratch, including custom low-level I/O, threading, and messaging components. Modern abstractions—such as servlet containers, ORM tools, or standard serialization libraries—were often unusable due to their latency and memory overhead.

Concurrency was particularly difficult to achieve and manage, as it required deep understanding and manual orchestration of multithreading. Scaling and ensuring high availability necessitated extensive custom engineering, since these concerns weren't natively supported by the underlying infrastructure.

Development environments were cumbersome to set up. Running a representative stack locally often required configuring clusters, setting up databases, caches, and coordination layers—resulting in long setup times and limited developer agility. DevOps workflows were similarly fragmented and labor-intensive, with multiple components needing independent provisioning and maintenance.

Performance Challenges from Data Tier Separation

Traditional architectures that separated compute and data tiers introduced performance bottlenecks that were difficult to overcome. Bid processing logic typically required fetching large and variable datasets from remote storage systems. This I/O introduced unpredictable latency and reduced overall throughput.

Even with caching in place, persistent storage remained a bottleneck for operations requiring durability or recovery. Java-based systems also faced frequent garbage collection pauses, largely driven by serialization/deserialization overhead, which further disrupted real-time responsiveness.

High Cost

The combined impact of architectural complexity and data movement inefficiencies significantly inflated development and infrastructure costs. Teams had to allocate considerable engineering effort to solve infrastructure problems before they could focus on business logic.

From an infrastructure standpoint, systems often required high-end hardware, large clusters, and costly third-party components just to meet baseline performance requirements. For example, setting up a distributed NoSQL store like Apache Cassandra required three to five dedicated servers, even for development or staging environments.

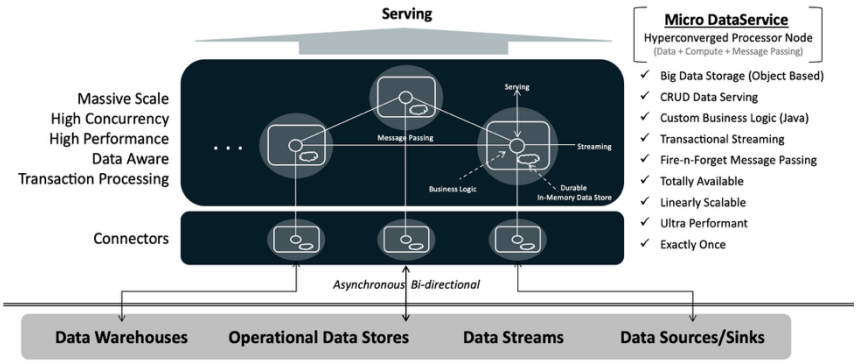
Case Study

Real-Time Ad Bidding

INTRODUCING RUMI™

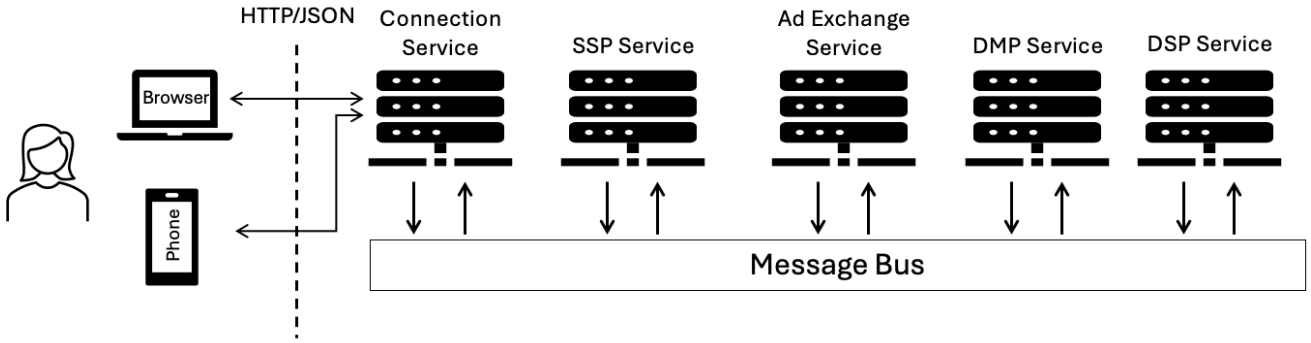
Rumi™ is a high-performance platform designed specifically for real-time, data and performance intensive applications like ad bidding. Its mission is to simplify the development, deployment, and scaling of real-time systems while enabling extreme performance and ensuring enterprise-grade reliability and resilience.

The core problem with traditional architectures is the separation of compute and data tiers across a network. This separation introduces latency and complexity, making it difficult to meet the strict performance and scalability demands of real-time systems. Even with high-end databases or optimized network layers, the cost and delay of fetching data on demand remains a bottleneck.



Rumi™ solves this by eliminating the network between compute and data. It brings both together into a single, hyper-converged software node where each node acts as both a first-class compute engine and a data store. Application logic runs directly against in-memory state, removing the need for external I/O during processing. This design enables Rumi™ to deliver high throughput, ultra-low latency, and built-in fault tolerance. Unlike platforms that focus solely on accelerating compute or data, Rumi™ accelerates both, allowing systems to process more data, faster, with fewer moving parts and without compromise.

THE RUMI™ BASED RTB ARCHITECTURE



The real-time bidding solution was implemented as a single, end-to-end system that interfaces with external clients over standard HTTP/REST/JSON APIs. Internally, it consists of four micro-dataservices, each communicating over a high-speed message bus using Rumi's efficient binary protocol. Every micro-dataservice is independently replicated for high availability and partitioned for horizontal scalability.

The key micro-dataservices include:

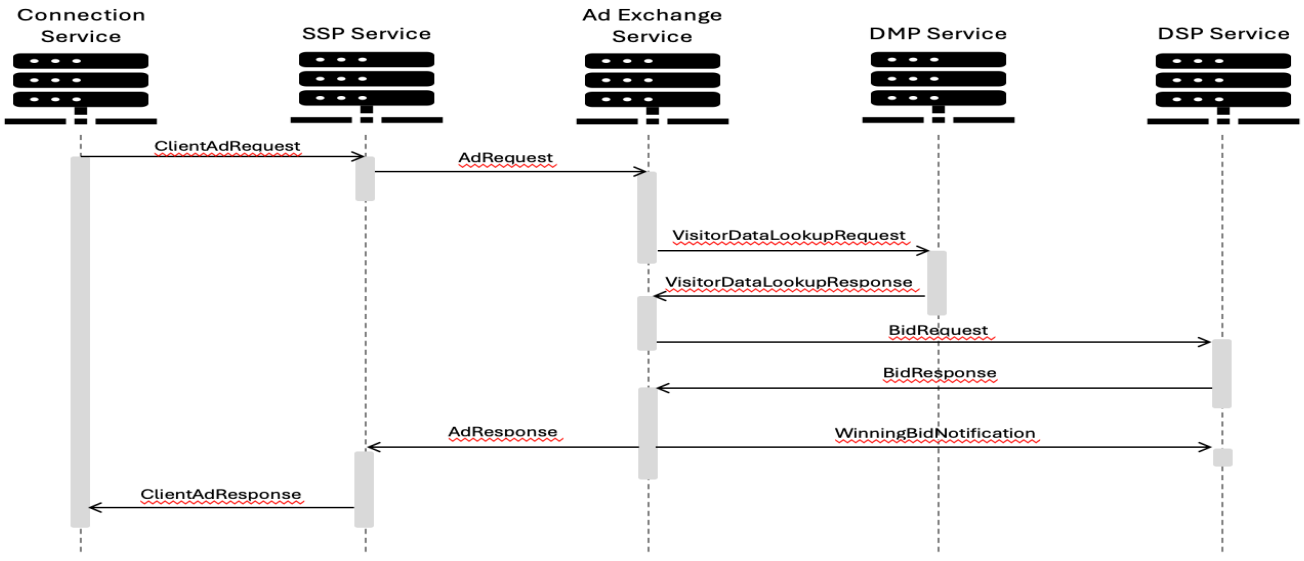
- **SSP (Supply-Side Platform)**
  Loads user behavior and interest data entirely in-memory. It receives incoming ad requests, enriches them with user and publisher context (e.g., content keywords), and forwards them to the ad exchange.

- **Ad Exchange:**
  Orchestrates the bidding process. It augments bid requests using data from the DMP, evaluates pre-cached bids when applicable, and dispatches bid requests to the DSP.

- **DMP (Data Management Platform):**
  Maintains hundreds of millions of user tracking records in-memory. This allows for ultra-fast lookups without the latency of traditional storage systems.

- **DSP (Demand-Side Platform):**
  Manages multiple advertisers, including account data and campaign logic. By co-locating the DSP and ad exchange within the same Rumi-based environment, the bidding logic is greatly simplified and execution latency minimized.

The diagram below shows the main message flow that drives the ad bidding process. From the moment an ad request is received to the point a response is delivered, the system performs eight internal message hops. After the auction is concluded, a win notification is sent to the DSP, which is responsible for managing multiple ad campaigns.



PERFORMANCE TESTING & RESULTS

The Rumi-based RTB system was deployed on three physical servers, each with dual 10-core 3GHz CPUs and 96 GB RAM. Internal messaging used Rumi's high-speed messaging backbone. Each cluster member of each of the services was deployed on separate server instances hot-replicated over the network, and a dedicated driver micro-dataservice was used to simulate incoming ad traffic.

TEST SCENARIO

- The DSP was configured to manage 1,000 active campaigns
- 100,000 ad requests were sent at a sustained rate of 1,000 requests per second.
- Each request triggered:
  - A read and write operation in the DMP's in-memory state.
  - A full search across all 1,000 campaigns in the DSP to identify bid candidates.
- Midway through the test, the primary ad exchange instance was deliberately terminated to validate failover behavior.
- The system was expected to preserve all in-flight data, maintain message continuity with minimal disruption and ensure zero message and data loss across failures

OBSERVATIONS

The table below summarizes the processing and messaging latencies for each step in the ad bidding flow. All timings are measured in microseconds (µs):

Step	Time (µs)
SSP Request	62
Ad Exchange → DMP	40
DMP Visitor Lookup	24
Ad Exchange → DMP	33
DSP Bid Response	463
Ad Exchange → SSP	26
SSP → Test Driver	30

Despite involving 8 internal message hops, the average end-to-end request/response time remained around **1.4 milliseconds**. The DSP was the most computationally intensive component due to its full-campaign evaluation logic, which dominated processing time.

Failover Performance

When the primary ad exchange instance was killed, failover to the backup completed within **1 second**. During this window, a temporary spike in request latency was observed:

- Maximum request latency reached **900 ms**.
- Average latency during this 1-second window was **68 ms**.
- A worst-case estimate suggests that **~7% of requests (73 out of 1,000)** may have temporarily breached the 100ms SLA during failover.

However, **no data was lost**. Rumi's built-in message durability ensured all affected messages were successfully redelivered. This made it straightforward to reconcile expired impressions and maintain system correctness.

CONCLUSION

Rumi is purpose-built for real-time, data-intensive systems like ad bidding—where ultra-low latency, high throughput, and fault tolerance are non-negotiable. By unifying compute and data into a single software node, Rumi eliminates the traditional network boundary that hinders performance, reliability, and scalability in legacy architectures. This case study demonstrated how a third-party team was able to build an end-to-end RTB system using Rumi's micro-dataservice model. The system achieved average bid resolution times of ~1.4 milliseconds across eight message hops, maintained seamless failover during node failures, and delivered guaranteed message delivery with zero data loss. Importantly, the platform simplified what would traditionally require extensive low-level engineering. Developers did not need to build custom concurrency, storage reliability, or failover mechanisms—Rumi handled these non-functional concerns out of the box, allowing teams to focus entirely on business logic and differentiation. By reducing complexity, increasing developer velocity, and enabling extreme performance, Rumi redefines how high-performance systems are built. It stands out not just as an infrastructure solution, but as a foundational layer for the next generation of mission-critical, real-time applications.