X Platform™

Architecting and Building a Real-Time Ad Bidding System

White Paper

by Igor Mihaljevic

Table of Contents

About Neeve Research

Neeve Research offers the X Platform™, an industry first, memory-oriented, multi-agent application platform for I/O intensive, message driven applications. The X Platform™ catalyzes enterprise agility by allowing developers to write plumbing-free business logic on a platform that seamlessly injects fault tolerance with zero data loss and near linear horizontal scalability while exhibiting extreme performance characteristics. The X Platform™ pioneers the use of memory-oriented computing, advanced messaging, and decoupled enterprise data management to enable no-compromise computing from a performance, scalability, agility, and reliability standpoint. The X Platform™ is running mission critical applications at Fortune 500 securities trading, resort, and gaming corporations. For additional information visit: http://www.neeveresearch.com.

About Kode41

Kode41 is a software development vendor with a project management office in San Francisco, CA and an engineering office in Novi Sad, Serbia. Kode41 specializes in the development of front-end (AngularJS, ReactJS) and back-end (Java, Python) components for complex software systems. For more information, visit *http://kode41.com/* or contact Kode41 at *info@kode41.com*.

Introduction

In the Kode41 portfolio, there are several projects comprising real-time ad exchanges and trading platforms for ad campaigns, built on a Java VM. The goal of this white paper is to give an introduction on ad bidding systems, our experiences developing them in Java, problems encountered, and how we are solving them with our newest design approach using X Platform™ by Neeve Research LLC [1].

Real-Time Ad Bidding: A Brief Introduction

Real-time Bidding (RTB) is a way of transacting media that allows an individual ad impression to be put up for bid in real-time. This is done through a programmatic on-the-spot auction, which is similar to how financial markets operate. RTB allows for Addressable Advertising; the ability to serve ads to consumers directly based on their demographic, psychographic, or behavioral attributes [2].

To get an overview on today's online advertisement placement, we will observe what happens in the typical ad serving system when an end-user is browsing and the page loads with a banner ad placeholder.

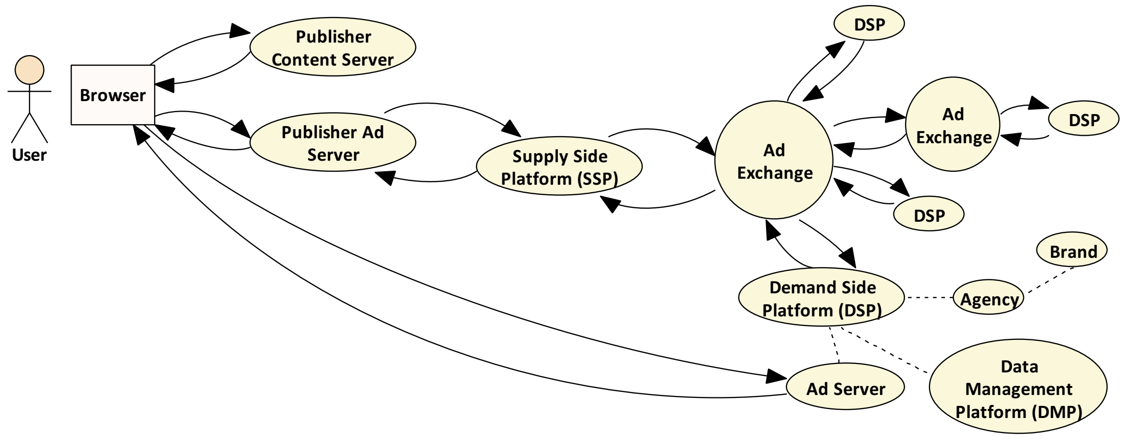

Illustration 1 Real-Time Ad Serving System

When the user's page loads, it contains an URL through which the ad should be retrieved. This URL points to the publisher's ad server. The ad server may define rules on which ad is to be served. For instance, if there is reserved advertisement space that an advertiser bought directly from the publisher, the ad will be served directly from the ad server and the process ends. If the ad space is not reserved, the ad server may contact the Supply-Side Platform (SSP) and offer to sell the ad space. The SSP may hold some data on user's behavior and interests to aid with the targeted advertisement. The SSP sends an ad request to the ad exchange that is appended with any additional useful information about the user and the publisher, such as content keywords.

The ad exchange, on receiving the ad request, search the Data Management Platform (DMP) for any information about the visitor, and then auctions this ad opportunity. The ad exchange will send bid requests to potential buyers, along with any available visitor information. The buyers are Demand-Side Platforms or other ad exchanges. The bidders may also pre-cache bids in bulk on the ad exchange. This works like setting an automatic buy/sell on a stock exchanges. The trade will execute when certain conditions are met. Conditions may include data about the end-user such as interests, age, or data on the publisher's current served content such as keywords. Bidders typically must respond within half of total time limit for ad exchange response.

Once the ad exchange receives the responses it does the following:

1. Determines the winner.
2. Debits the winner's account for price of bid.
3. Sends the win notification to the bidder.
4. Sends a response to the SSP with an URL for retrieving the ad. The ad exchange may get this URL either initially with the response of bid request, or in response to a win notification sent to the bidder.

The SSP passes the ad URL to the publisher's ad server and the ad server notifies the user's browser on how to retrieve the ad stored on the advertiser's ad server.

For small publishers that use a third party SSP like Google Ad Sense, the publisher ad server may not exist. An ad request may be sent directly from the browser to the SSP.

Problems with Prior Solutions

As is evident from the very nature of ad bidding, performance, reliability and scalability are all very important.

- Performance needs to be low enough that an ad bid is resolved within a low enogh time that is imperceptible to the user. Additionally, there are several spaces on the browser page that are concurrently being bid for and, therefore, the system needs to

support high concurrency and throughputs while still ensuring a low, imperceptible bid resolution latency.
- Reliability is key to ensure that there are no requests or data is lost on failures to ensure a consistent user experience and accurate bid resolution. This reliability needs to be encured not only in the context of a single component but also across various components that are interacting in the overall bid resolution flow.
- Finally, such systems exhibit spiky load profiles and therefore need to possess mechanisms to scale horizontally to absorb higher loads.

Achieving these non-functional SLAs in totaliy is very complex and requires complex programming at a system level. This complexity results in an overall increase in the risk profile of the system and compromises functional agility by taking time away from the functional aspects of such systems.

Some of the key problem areas with prior implemnetations of ad bidding systems include

- **Build Everything from Scratch** – To achieve maximum performance, we had to develop highly customized low level I/O and multi-threading libraries. We could not use heavy Servlet containers to do HTTP, O/R mappers to model and manage data persistence, or bloated serialization libraries which put load on CPU and create garbage in-memory.
- **Concurrency was Difficult to Achieve** – We had to design and deal with multi-threading at low-level.
- **Scaling is Difficult to Achieve** – This comes from the complicated concurrency. Nothing is supported by the infrastructure, it all had to be written by us.
- **Reliability was Difficult to Achieve** – Replicating services for high availability required custom design and special planning.
- **Storage is the Bottleneck** – Reliability of operation, and integrity of data is based on persistent data storage. We have worked with several commonly available SQL and NoSQL systems, and the storage is always the point of contention. No matter how well we optimize our application this is a concern. We rely heavily on remote cache, but even using cache doesn't solve it, since some data must be written for recovery and other purposes.
- **Real-Time Computing and Garbage Collection Don't Work Well Together** – We had issues with Java GC pauses, which degrade performance of the system. Most of garbage was caused by serialization/deserialization.
- **Complicated Setup for Development** – Setting up the development environment was tedious, especially if developers needed a full stack on their box. With some of the

NoSQL software, it wasn't even possible to fully test everything without running a three-machine cluster. Also, every component needed setup and configuration.

- **Complicated DevOps** – We ended up with many different components to setup and configure. There was no infrastructure for deploying the application itself. Since many of the servers have different roles, this required different kinds of management – database servers, cache servers, application servers, etc.
- **Complicated Hardware and Network Topology from Project Start** – Today's organizations are very cost-aware. They need a way to reduce cost and scale as their business grows. Lowering the development and production cost by using developer workstations and commodity hardware was often difficult. For example, setting up an Apache Cassandra cluster for storage requires at least three servers and five servers is the recommended minimum.
- **Cost of Development** – We had greater engineer effort because we had to spend resources developing not just the business logic, but also all the infrastructure running it.

The New Approach: Message-Driven In-Memory Ad Bidding System on X Platform (TM)

For this new project, we selected The X Platform™ because it is a general purpose platform for hyper-distributed, multi-agent applications. X Platform based agents regularly process 100s of thousands of transactions per second in single to double digit microsecond level latencies with linear horizontal scalability and full recoverability with zero loss on process, machine, network and data center failures. All of this with centralized deployment management including configuration, command-n-control, monitoring, versioning, upgrade with zero service interruption and drill down capabilities for troubleshooting.

We found that the X Platform™ is powered by a key innovation that centers around the *use of in-memory computing in a unique manner that eliminates the boundaries between application logic, messaging, state management and transaction processing*. It integrates them in a manner that allowed us to author application logic, manage state and communicate with other agents using "plain old Java objects" in an easy, fully integrated and transactional manner without having to concern ourselves with the non-functional aspects of the applications. The platform allowed us to author application logic using Java, manage state using POJOs *assuming memory is durable* and communicate with other agents using POJOs in a *"fire-n-forget"* manner *assuming exactly once delivery semantics*. By eliminating the complexities of state management and messaging and making them completely technology agnostic, and allowing our development to focus exclusively on application logic, we foundusing this platform for an ad bidding engine had some key advantages.

By leveraging the The X Platform™ we architected the ad bidding product as a collection of micro applications as depicted in the illustration below.

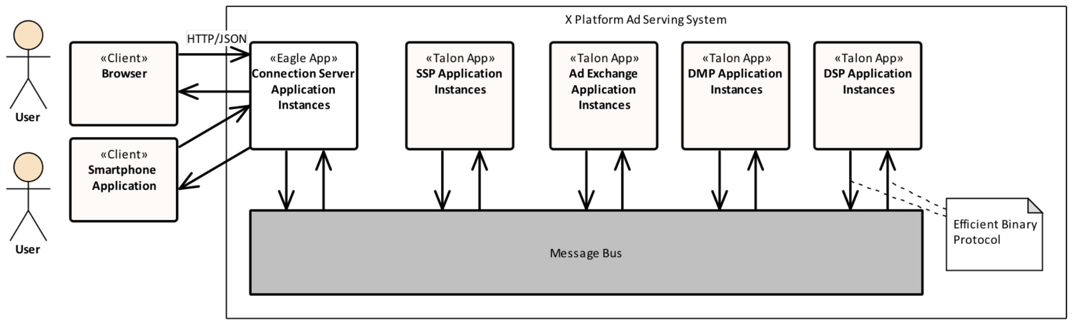
X Platform Ad Serving System
User
User
«Client» Browser
«Client» Smartphone Application
HTTP/JSON
«Eagle App» Connection Server Application Instances
«Talon App» SSP Application Instances
«Talon App» Ad Exchange Application Instances
«Talon App» DMP Application Instances
«Talon App» DSP Application Instances
Message Bus
Efficient Binary Protocol

Illustration 2 Ad Bidding System Built on The X Platform™

We decided to build a single end-to-end system which interfaces with external clients via HTTP/REST/JSON. It is composed of four micro applications that communicate with each other over a message bus. Internal communication is done with efficient a binary protocol provided by the platform. Each micro application is replicated for high availability and partitioned for horizontal scaling.

A brief description of each micro application is as follows:

- The SSP containing data on user's behavior and interests are now loaded in-memory. The SSP sends an ad request to the ad exchange that is appended with any additional useful information about the user and the publisher, such as content keywords.
- The ad exchange processes bid requests adding any available visitor information from the DMP, checks for pre-cached bids and notifies the DSP.
- The DMP containing user tracking data and hundreds of millions of entries is now loaded in-memory, with application state and is very fast.
- The DSP manages multiple ad buyers, who open accounts and create ad campaigns on the DSP. Since the ad exchange and DSP are now part of the same system, the bidding algorithm is simplified.

The following is the sequence diagram of the main message flow that facilitates the ad bidding. From the point of receiving an ad request from the client to the delivery of the response there are 8 message hops and a win notification that is sent to the DSP concluding the auction. A DSP is managing multiple ad campaigns.

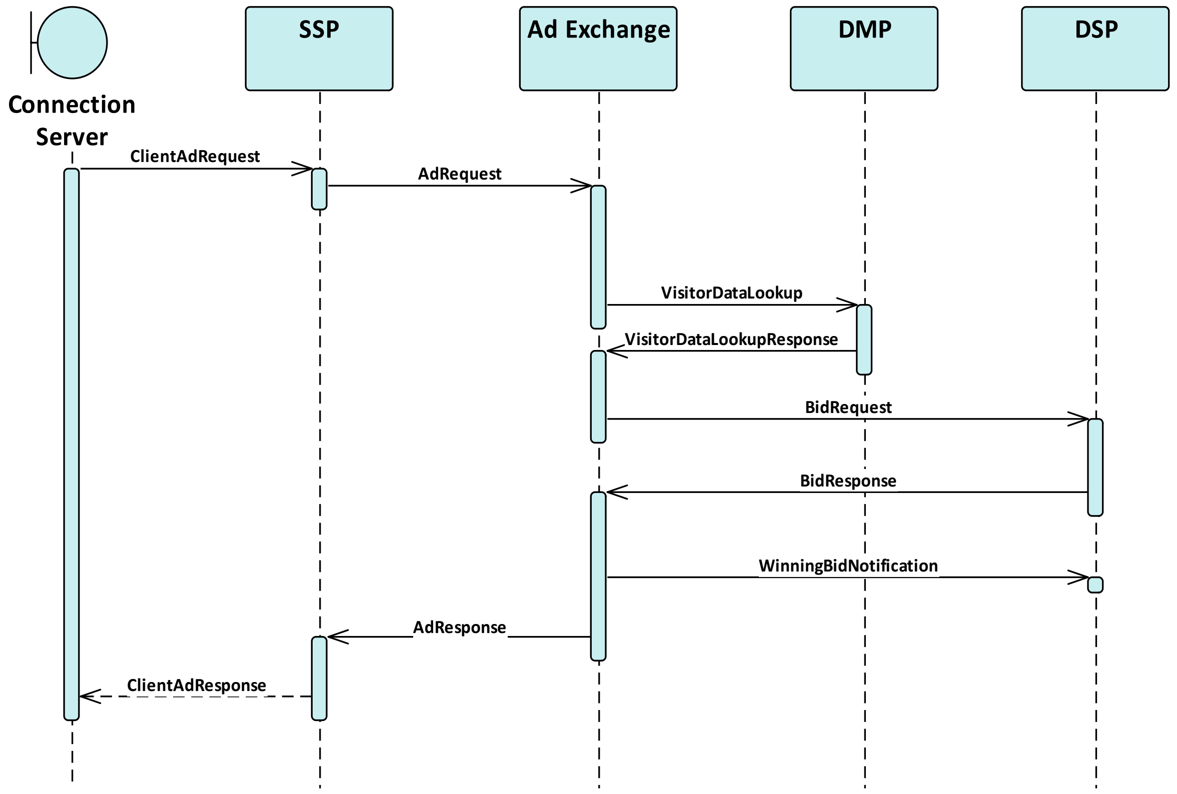

Illustration 3 Sequence Diagram of RTB Message Flow

Additionally, a copy of these micro applications can be found at: https://github.com/neeveresearch/nvx-app-ad-bidding-engine.git

Benchmarks

For testing performance, the system is deployed on 3 physical servers with dual 10 core CPU 3GHz and 96 GB RAM. The messaging provider is the X Platform™ Solace Messaging Router. The ad exchange application is deployed as two instances: a primary and a backup. The connection server is replaced by a driver micro application which simulates large traffic. Test runs as follows:

- DSP instance manages 1,000 campaigns.

- We are sending 100,000 ad requests at rate of 1,000 requests/s.
- DMP will do at least one read and one write in the state for each ad request.
- DSP will search all campaigns in the state for bid candidates on each ad request (since all campaigns are competing independently).
- At some point, we will intentionally kill the primary ad exchange instance.
- We should see that failover to backup will redeliver any messages that ended up on primary instance at the moment of failure with no data loss.
- We expect that node failure will cause negligible delay to the flow.

Our findings are shown below. The table shows the processing and messaging times in the message flow. All times are below 1 millisecond. The bottleneck is the DSP because it is computationally most complex. It searches all 1000 campaigns at every bid request.

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

Table 1 Processing times in the message flow

The chart below shows the average for total ad request to response times for the entire duration of the test. It is around 1.4 milliseconds. When we add up times from the table above, we can see that they do not add up to 1.4 ms. There is slight overhead in the messaging provider due to the way it is configured in the lab. Meaning, there is a potential for even better performance with slight adjustment to the setup. We should also note that these times are for a message flow with 8 message hops. Given the industry norm, the achieved performance is outstanding even as it is now. The chart also reflects the spike up to 68 ms when we killed the ad exchange primary instance. The X Platform ™ provided up to 1 second failover (the time needed to redeliver any messages that went to failing node). The maximum request - response time was around 900 milliseconds.

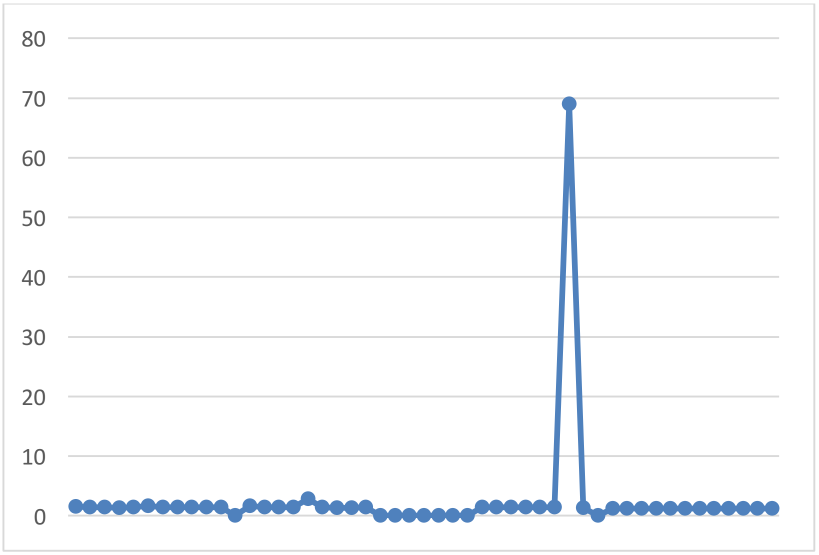

Chart 1 Average Ad Request – Response time (milliseconds)

Knowing that the duration is 1 second, that we have 1000 requests, the average is 68 milliseconds. The maximum latency is at 900 milliseconds. We can now calculate the worst case scenario for the number of requests that broke the service level agreement of 100ms to delivery. The worst case scenario would be to assume that all delayed responses are at the maximum of 900ms and that the all best response times are at the average of 1.4 ms. This gives us ~73 responses that are outside of SLA.

An instance failure caused a delay with up to ~7% (73 out of 1000) of responses outside of SLA, and only during 1 second. Compared to the total traffic, this is negligible. Furthermore, the guaranteed delivery ensured that we still get the delayed messages, making it very easy to take any corrective action and notify other applications of expired ad impressions.

Problems Revisited: How the X Platform™ Solved Them

Here is how problems with our previous project were overcome by leveraging the X Platform™:

- **No "Write from Scratch"** – There is no dealing with I/O, sockets or threads directly to achieve real-time performance. Serialization is built-in and highly efficient. A fast and reliable storage is built-in. The X Platform™ easily exceeds ad serving/bidding time constraints 100/50ms by orders of magnitude, without any additional designing and planning. The X Platform™ also provides a modeling language and a code generator. The data model and messages can be represented using XML, from which Java classes are generated. The X Platform™ left us only dealing with three aspects: configuration, pure business logic, and the data model.

- **Messaging is Simple** – Apart from configuring the topic/channels, the developer only deals with sending messages and writing handlers for them. Everything else is managed by the platform, and business logic is ignorant of any messaging mechanism internals.
- **Concurrency Model is Built In** – The X Platform™ programming model is like the actor model. This actor model covered most of our use cases. To plan for concurrency, we need to partition data in chunks. Chunks can be managed and accessed independently. The X Platform™ module called Talon compartmentalizes these chunks when the application state is being accessed, so that when there are new events, they will be processed in sequence. Access to application state is always thread-safe.
- **Horizontal Scaling was Simple** – If application state is well partitioned, Talon (TM) will provide as much concurrency as there are available resources and instances of the application.
- **Reliability is Simple** – Talon manages replication for high availability. For the most part, it is just about running additional instances of the same Talon application.
- **Storage is Fast** – No bottlenecks on write. Data is in memory, replicated for high availability. Access is fast. Events can be processed in microseconds, even when state needs update. State persistence is managed by X Platform automatically. Business logic need not do anything explicitly to persist data, apart from mutating application state.
- **No Remote Cache** – Application state is in-memory and in process, so access times are very fast.
- **No Garbage Collection Pauses** – By abiding by X Platform™ development guidelines, we managed to avoid garbage collection pauses and high CPU loads. X Platform provides object pooling, and can recycle object instances, thus reducing new memory allocations.
- **Easy Setup for Development** – Since there is (are) no database servers, no remote cache servers, or other components, it is very easy to set up the development box and get everything running on a single machine. It is also easy to test very complex systems on a single machine. When starting a new project, only a few steps are needed – set up a license file and get the talon example app running. From there we just implement the business logic, and add any configuration needed to run it.
- **Simplified DevOps** – For the most part, our servers look identical now. They all run the same software, and are managed in the same way. Deployments are simplified by

using Robin Server (TM) which acts as a container and remote deployment tool for X Applications. The X Platform is highly introspective because it provides operation statistics, both for the business logic and for internal X Platform™ components involved in processing an event. There is a graphical representation with Lumino (TM) server, so that system can be monitored from a web browser. This is especially important for our speed requirements – to be able to measure message flow timing, and profile for possible contention points.

- **Simplified Hardware and Network Topology** – It is possible to get a basic system running with less hardware, which is of interest to startup businesses that don't have a big cloud infrastructure.
- **Cost of Development is Reduced** – Since we focus on business logic only, and not on the platform that runs it, we can reduce development costs and delivery times.

Future Considerations

For our next deployment, we plan to split the system into separate products and expose their API through the standardized OpenRTB protocol by iab. [2]. This means separating each micro application and then interfacing them via HTTP. Doing this enables us to interface with third party systems. For example, the DSP alone could participate in bidding on Google's DoubleClick (TM). Exposing ad exchange as a standalone product enables auctioning of the ad space to third party bidding systems. Interfacing incoming traffic via HTTP is possible by using either a module in the X Platform™ referred to as the Eagle server or writing a custom connection server and implementing bus protocol. Outbound HTTP is possible via custom HTTP client or by mechanism provided by The X Platform™ to send message as HTTP directly from micro application.

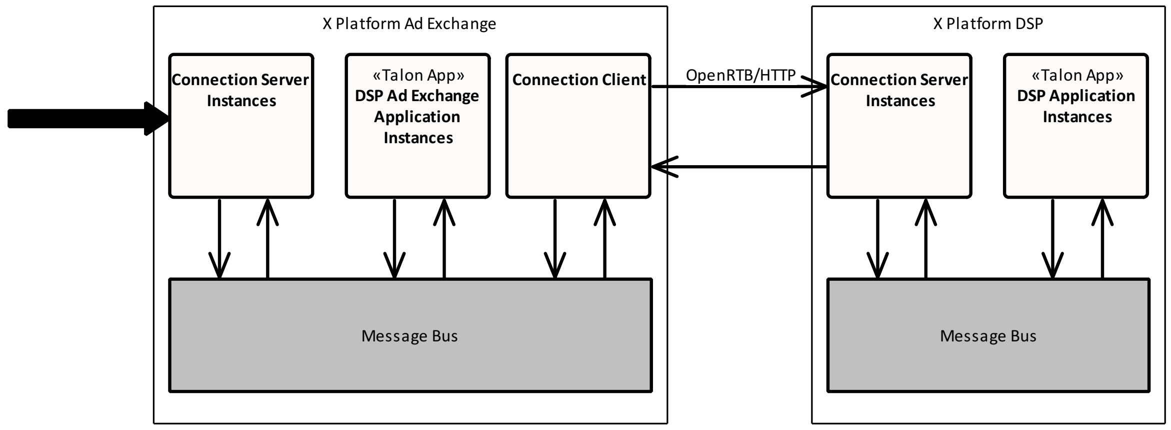
X Platform Ad Exchange
Connection Server Instances
«Talon App» DSP Ad Exchange Application Instances
Connection Client
OpenRTB/HTTP
X Platform DSP
Connection Server Instances
«Talon App» DSP Application Instances
Message Bus
Message Bus

Illustration 4 Ad Exchange and DSP as Separate Platforms

Conclusion

We find that event driven processing is a very convenient solution for ad serving systems. The combination of local in-memory transactional data store, message flow-based processing, zero garbage collection and performant storage and messaging solved our ad bidding timing requirements. The X Platform™ provided all of these mechanisms out of the box that made is very easy to implement a highly performant and resilient ad bidding system.

References

1. X Platform™ [http://www.neeveresearch.com/]
2. OpenRTB Protocol [https://www.iab.com/guidelines/real-time-bidding-rtb-project/]