DSP Framework
Architecture Notebook

1 Purpose

This document describes the philosophy, decisions, constraints, justifications, significant elements, and any other overarching aspects of the system that shape the design and implementation.

2 Architectural goals and philosophy

The DSP Framework is intended to provide a comprehensive Demand Side Platform (DSP) implementation that will integrate with public and private exchanges, as well as mediated platforms to provide recommendations for creatives to satisfy their requests. It provides an open architecture with the following objectives:

- Extensibility to new exchanges - the addition of new exchanges with no changes required for the overall architecture
- Extensible and tailorable augmentations of the requests and replies to accommodate differences in the content of the requests and replies required by the exchanges, and
- Extensible and tailorable filters to assist in efficient filtration of creatives based on the requests and augmentation data

- Efficiency and scalability to allow for processing billions of requests per day.

3 Assumptions and dependencies

The following assumptions are agreed to for the execution of this project:

Assumptions
1. The first release will integrate with the OpenX exchange.
2. OCI will validate interface integration with OpenX's Sandbox.
3. All integration testing will be performed using Voltari resources.
4. Exchange communicates via HTTP using Protocol Buffer format bid requests and bid responses. Voltari will investigate whether they will move from JSON to another mechanism.
5. The Ad Server that will be used for the first release will be mOcean.
6. Interfaces for Bidding Agent will be REST using JSON. Collocation of OCI & Voltari code to be investigated.
7. Flume will be used in the first release of the system in support of the logging approach.
8. Support for configurable Augmentation will be provided by the framework. Augmentation will include data provided by Neustar, WURFL and the Voltari Reverse IP Geocoder. The interfaces to Neustar and the Geocoder will be initially implemented using the existing REST interfaces.
9. OCI will provide a scoreboard to monitor and display current build and test status.
10. OCI will supply configuration management of software engineering work artifacts while the development effort is in progress, but will transfer that responsibility to Voltari once the effort is completed.
11. A UI configuration interface be needed for the Router or just use Config files eventually but not in Phase I.
12. Exchange Integration will need to be able to eventually handle OpenRTB 1.0, 2.0 and 2.1 and translations between them. The design must accommodate eventual support for these interfaces.
13. The run-time platform for the first release will be Linux - CentOS 6.3.

4 Architecturally significant requirements

- **The DSP Router shall support up to 1 billion requests per day.** [An additional goal should be set on the number of bid responses we expect to be able to accommodate as well – as this requires end-to-end processing of the request, its evaluation and the submission of the bid response.]
- The system shall log all DSP events including: augmented bid requests, augmented bid responses, auction results (win/loss), financial data, click, impression, conversion, scores, weights, indices, and filters applied.
- The system shall ensure all real-time DSP events are available for analysis within 20 seconds.
- The system shall ensure all non-real-time DSP events are available for analysis within 20 minutes.

5 Decisions, constraints, and justifications

- JSON will be used as the internal communication format for data between the DSP Router and the Bidding Agent for the early phase of the project. This format provides flexibility for a variety of request and reply content to allow for extensibility to different exchanges and mediation platforms. For later releases we may consider moving to a binary format such as Protocol Buffers to improve efficiency.

6 Architectural Mechanisms

This section provides a description of the architectural mechanisms used within the DSP Framework. This section will be refined as specific mechanisms are identified and integrated into the DSP Framework.

7 Key abstractions

- Ad Request – A message provided by a Mediation Platform requesting an Ad from the DSPL.
- Ad Server - An application/server that helps publishers manage multiple advertising requests and advertisers manage multiple advertising campaigns.
- Auction – The workflow that occurs in response to a Bid Request and resulting in a Bid Response.
- Bid Request – A message provided by the Exchange requesting a Bid from the DSP.
- Bid Response – A message provided to the Exchange by the DSP supplying bids for a received Bid Request.
- Creative – A banner or other form of created advertising.
- Demand Side Platform (DSP) – A system that allows buyers of digital advertising to manage ad exchange and data exchange accounts though a common interface. The architecture of this application implements the DSP functionality by responds to Bid Requests and Ad Requests with a corresponding Response as part of real time bidding arrangement.
- Mediation Platform – A source of Ad Requests for which there are a predefined set of Creatives available.
- Request Augmentation – Adding or refining information in a Bid Request based on data available to the DSP upon receipt of the Bid Request.

- Bidding Agent Filter – A component that identifies whether existing Creatives are eligible for the Bid Request. Filtering for the Bidding Agent will be performed within the DSP Router process as part of the request processing flow.
- RT Cache – A component used by the DSP which contains data used in refining the filters, augmentations and bid responses within the DSP.

8 Architectural framework

Patterns

As architectural patterns are identified and applied in the architecture they will be documented in this section.

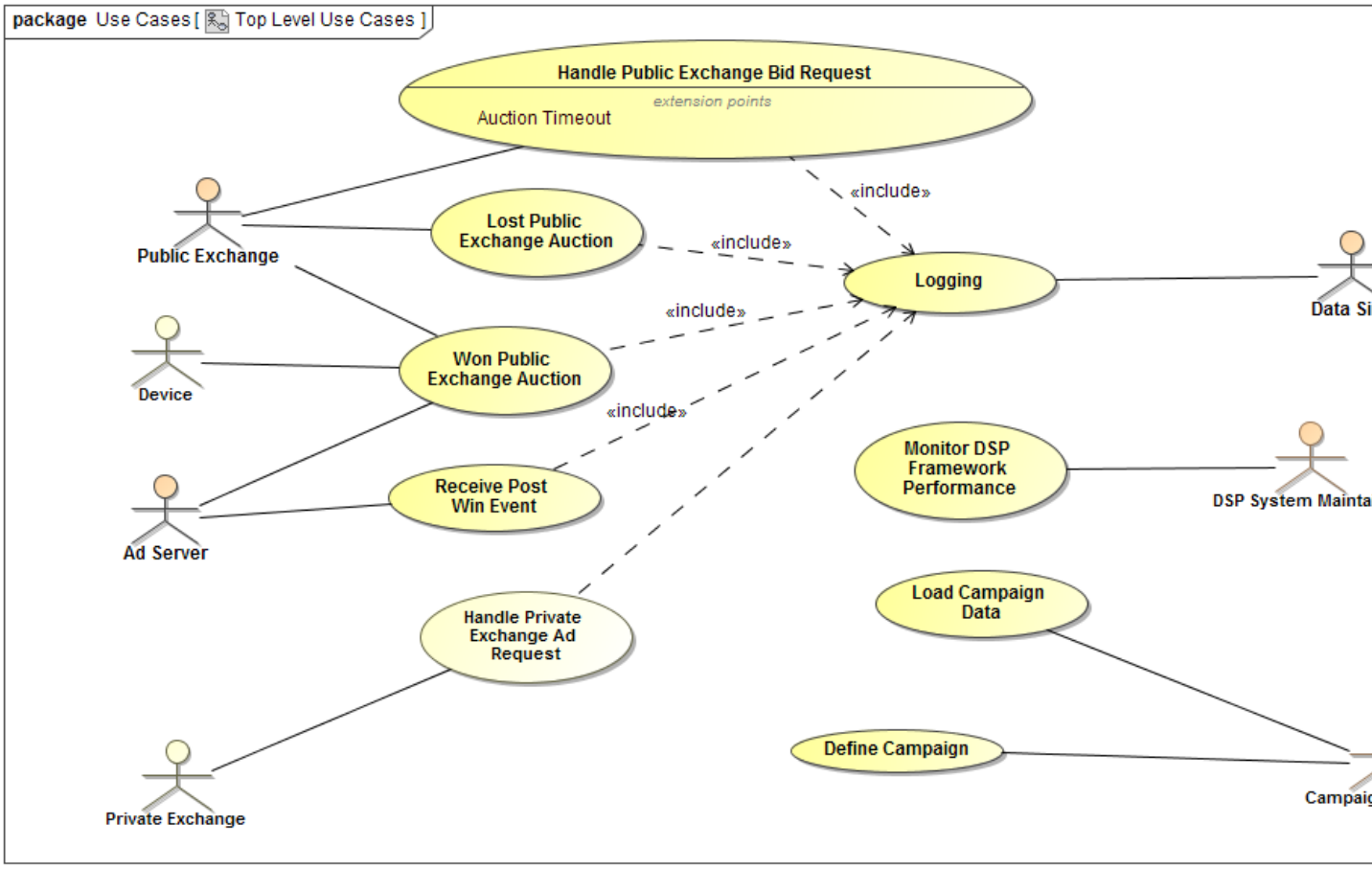
Service Configurator – This design pattern allows an application to link and unlink component implementations at run-time, reducing the need for modification, recompilation or static linking of applications. This approach will be used by the DSP router to support dynamic configuration of the DSP Router Exchange Handlers and Augmentations.

Chain of Responsibility – Request processing in the DSP Router is based on the Chain of Responsibility behavioral pattern [GOF95]. In this pattern, there is a sequence of objects (a chain) that break up the processing to a sequence of handlers. In the DSP Router, these handlers are implemented as derived classes of the BidStage class. This pattern is described in greater detail later.

9 Architectural views

The architecture description currently consists of a Use Case view, and a Logical View. The Use Case view consists of a Use Case diagram which provides an overview of the high level use cases and actors within the DSP Framework. These use cases are described with a nominal flow in the current revision of this document.

Use Case View



Handle Public Exchange Bid Request

Preconditions:

1. Connection established with the Exchange
2. Connection established with the Bidding Agent

Extension Points:

- If the Bidding Agent does not respond in a timely manner as defined by the response requirements of the Exchange, an Auction Timeout will occur. The Auction timeout will require the logging of the issue with the Bidding Agent and a 'No Bid' response to the Exchange.

Steps:

1. Exchange Connector Receives a Bid Request from the Exchange
    1. Exchanges (public, private & mediated) include: Admeld, AdX, MobClix, MoPub, Smaato, Nexage
2. DSP Router parses the BidRequest object (see rtbkit/openrtb/openrtb_parsing)

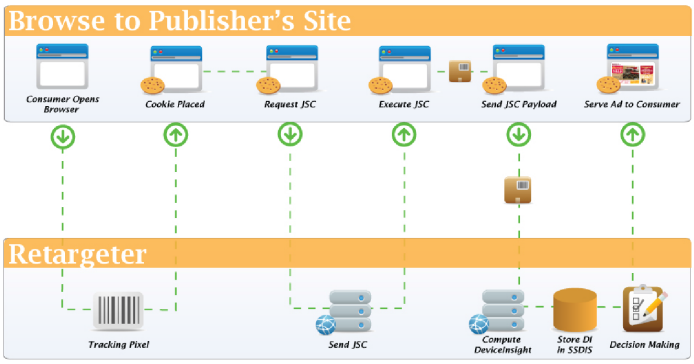
    1. Bid requests may be in OpenRTB (1.0, 2.0 or 2.1) or OpenX format.
3. DSP Router Augments Bid Request using the following services:
    - Neustar - Provides reverse IP lookup. (updates every night background)
    - DeviceAtlas - Device Mapping (Updates once per quarter)
4. DSP Router determines which of the active creatives are eligible for the auction based on a set of filters, the creatives obtained from the RT Cache and the accumulated counts and spend on each of the creatives.
5. DSP Router Transforms Bid Request to Voltari (Common) Interface (JSON processing lib)
6. DSP Router sends augmented Bid Request to Voltari Bidding Agent with Filtered Eligible Creatives (when number of eligible creatives > 0).
7. Bidding Agent (aka Ad-Reco) receives the request
8. Bidding Agent performs filtering as needed (based on creative status, CPM, budget, etc).
9. Bidding Agent does dataset creation, scoring, index/weighting. The dataset creation will use data from the cache. May potentially log the scores, weights and indices. The cache will be updated from an offline process.
10. Bidding Agent performs bid calculation. This may access the cache.
11. Bidding Agent builds the response. The request creation will require data from the cache.
12. Bidding Agent submits Bid Response to the DSP Router (this consists of a single Creative)
    1. Support a timeout in the Router so that if the Bidding Agent does not respond in a timely manner, we can respond to the exchange. 'tmax' in Bid Request provides the max time for the response. If there are exchanges that do not require a response, this will be implemented as a configurable feature.
    2. For OpenX, bidders are given 125 ms to submit a bid response. OpenX suggests that responses should be <= 1k in size – although this is not a hard requirement. Different exchanges will have their own response time requirements.
    3. For future builds may need to support Load Shedding to accommodate degradation under load. [See RTB Kit https://github.com/rtbkit/rtbkit/wiki/Load-shedding]
13. DSP Router Receives Bidding Agent Bid Response
14. DSP Router Augments and Transforms the Bid Response to Exchange format.
    1. One potential augmentation might be a Tracking Pixel.
    2. OpenX allows you to provide the URL for the ad in the BidResponse. Upon a win, the exchange will load that URL in the user's browser. OpenX supports a set of macros which can be added to each Bid in the bid response. These are substituted by the exchange dynamically prior to sending the ad to the end-user. This URL will be augmented to ensure that the Router is notified of the impression served.
15. DSP Router logs Correlated Bid Request and Response with Logger
16. DSP Router transforms the Bid Response to the Exchange format. OpenX has their own format. For future releases will likely need to support OpenRTB version 1.0, 2.0, 2.1.
17. DSP Router Sends Bid Response to Exchange (for OpenX a BidResponse is required, otherwise they will penalize the bidder with a reduced number of BidRequests). Where there are multiple placements, a CPM bid or no bid response must be provided for at least one of the matching placements). Bidders may also include multiple bids in the Bid Response.
18. Once the need for the Bid Request and Response objects has gone away, the DSP Router cleans up these resources. Portions of the Bid Request are needed until the exchange auction is completed and we have either won or lost in order to supply this information to the Event Correlator.

Apply AdTruth Tracking

Preconditions:

Steps:

1. The device issues a click request to the DSP Router
2. The DSP Router looks up the "auction_id" and ad truth payload from the Ad Truth Service
3. The ad truth payload is used to generate an ad truth id which is used for logging (see the DeviceInsightController (DeviceInsightController.groovy)
4. Either a redirect (302) is returned or if javascript is enabled, the javascript is returned to the device which will result in logging



Logging

DSP Platform provides the following information to the Logger:

- Augmented Bid Request
- Bid Response from Reco
- Augmented Bid Response (Provided to the Exchange)
- Auction Results (Win/Loss)
- Financial data (Might have regulatory requirements)
- Optional: Click, Impression, Conversion
- Scores, Weights, Indices
- Filters Applied (Beyond Phase I) including data indicating the reason behind a filtering decision. TODO:Utilize Serving Status table to help us log the right codes for reasoning for filtering. To be handled in a follow-on phase. Will add more codes to include case where Bidding Agent recommends No Bid.

Won Public Exchange Auction

1. DSP Router receives auction results from the Exchange
    1. May not receive a notification of loss, so we will need to support timeout updates for the auction to assume a loss.
    2. OpenX defines an AuctionResults message which provides notification of a Win and a Loss. If a loss, it provides a reason for the loss (insufficient bid, or filtered out – e.g., due to advertiser being blocked). They also supply the bid price of the winning bid. For a win we are notified of the actual price paid.
2. If the auction was lost, the Auction result is logged – including the reason for the loss if it is provided. No further processing of the loss notification is required.

3. When the auction notifies of a win, the DSP Router notifies the Event Correlator of the Bid Request and Win and includes relevant portions of the originating Bid Request data.
4. DSP Router logs Win Message with the correlated Event Request
    1. <Logging of the financial data might be constrained by regulatory issues and others that limit our ability to log it. This might require encryption of the financial data.>
5. Event Correlator Receives Event from the Impression Pixel Fire
    1. Post Auction Events are below:
        1. Impression
        2. Click (Can be a conversion)
6. Event Correlator redirects to URL supplied in the ad markup. Tracking pixel URL will contain URL for redirect.
7. Event Correlator correlates the event with the original bid request
8. Event Correlator Logs the correlated event.
9. The Event Correlator will notify the Count Correlator of the impression/click for the winning creative.
10. Event Correlator will update the cache with the count on the winning creative for impression and click.

Auction Timeout

The DSP Router is responsible for monitoring the auctions which are initiated upon receive of a Bid Request from an exchange. It may be necessary for the DSP Router to submit a No Bid response in some circumstances - Mediated Exchanges may require this response. Does a Public Exchange require a response in the case of a No Bid response?

1. DSP Router waits for duration based on auction time parameters
2. Upon timeout with no Bid Response from Bidding Agent the DSP Router responds with No Bid response to Exchange
3. DSP Router logs timeout and No Bid Response

Configure DSP

The DSP Router will support configuration parameters to be specified at startup and run-time to influence the behavior and performance of the DSP Router. The initial list of configuration parameters includes:

- Exchange connection info (RTB, Mediation, etc.)
- Augmentation configuration
- Optional data inclusion for Bid Requests
- Bidding Agent URL
- Flume parameters (file system locations, when to roll the file, # files to keep)
- Performance monitoring parameters (what to collect, etc.)
- Notification targets (email, text)
- Test mode

Handle Private Exchange Ad Request

Preconditions:

1. Connection established with the Exchange
2. Connection established with the Bidding Agent

Extension Points:

- If the Bidding Agent does not respond in a timely manner as defined by the response requirements of the Exchange, an Auction Timeout will occur. The Auction timeout will require the logging of the issue with

the Bidding Agent and a 'No Bid' response to the Exchange.

Steps:

1. Exchange Connector Receives a Bid Request from the Exchange
    1. Exchanges (public, private & mediated) include: Mocospace & Trion
2. DSP Router parses the AdRequest object
    1. Bid requests will follow the Mocean interface specification.
3. DSP Router Augments Bid Request using the following services:
    - Should all be the same as in Public Exchange use case.
    - Initial pricing stage unneeded in this use case.
4. Creative Filtering will be handled the same as in Public Exchange use case.
    1. Exchange ID / Zone relationship may impact this implementation.
5. DSP Router Transforms Bid Request to Voltari (Common) Interface (JSON processing lib)
    1. No changes in the interface required.
6. DSP Router sends augmented Bid Request to Voltari Bidding Agent with Filtered Eligible Creatives (when number of eligible creatives > 0).
7. Bidding Agent (aka Ad-Reco) receives the request
8. Bidding Agent performs filtering as needed (based on creative status, CPM, budget, etc).
9. Bidding Agent does dataset creation, scoring, index/weighting. The dataset creation will use data from the cache. May potentially log the scores, weights and indices. The cache will be updated from an offline process.
10. Bidding Agent performs bid calculation. This may access the cache.
11. Bidding Agent builds the response. The request creation will require data from the cache.
12. Bidding Agent submits Bid Response to the DSP Router (this consists of a single Creative)
    1. No final pricing data will be provided.
    2. Support a timeout in the Router so that if the Bidding Agent does not respond in a timely manner, we can respond to the exchange. 'tmax' in Bid Request provides the max time for the response. If there are exchanges that do not require a response, this will be implemented as a configurable feature.
13. DSP Router Receives Bidding Agent Bid Response
14. DSP Router Augments and Transforms the Bid Response to Exchange format requested in the AdRequest.
    1. There is a default format which could be overridden on request.
    2. One potential augmentation might be a Tracking Pixel.
    3. OpenX allows you to provide the URL for the ad in the BidResponse. Upon an impression, the exchange will load that URL in the user's browser. OpenX supports a set of macros which can be added to each Bid in the bid response. These are substituted by the exchange dynamically prior to sending the ad to the end-user. This URL will be augmented to ensure that the Router is notified of the impression served.
15. DSP Router logs Correlated Ad Request and Response with Logger
16. DSP Router transforms the Ad Response to the requested format. XML, HTML, JSON as defined in the Mocean specification.
17. DSP Router Sends Bid Response to Exchange. Assume only one ad requested per exchange request received.

Logical View

Top Level Architecture

The following layers will be used to manage the dependencies within the design and development of the DSP Framework to ensure maintainability and extensibility.

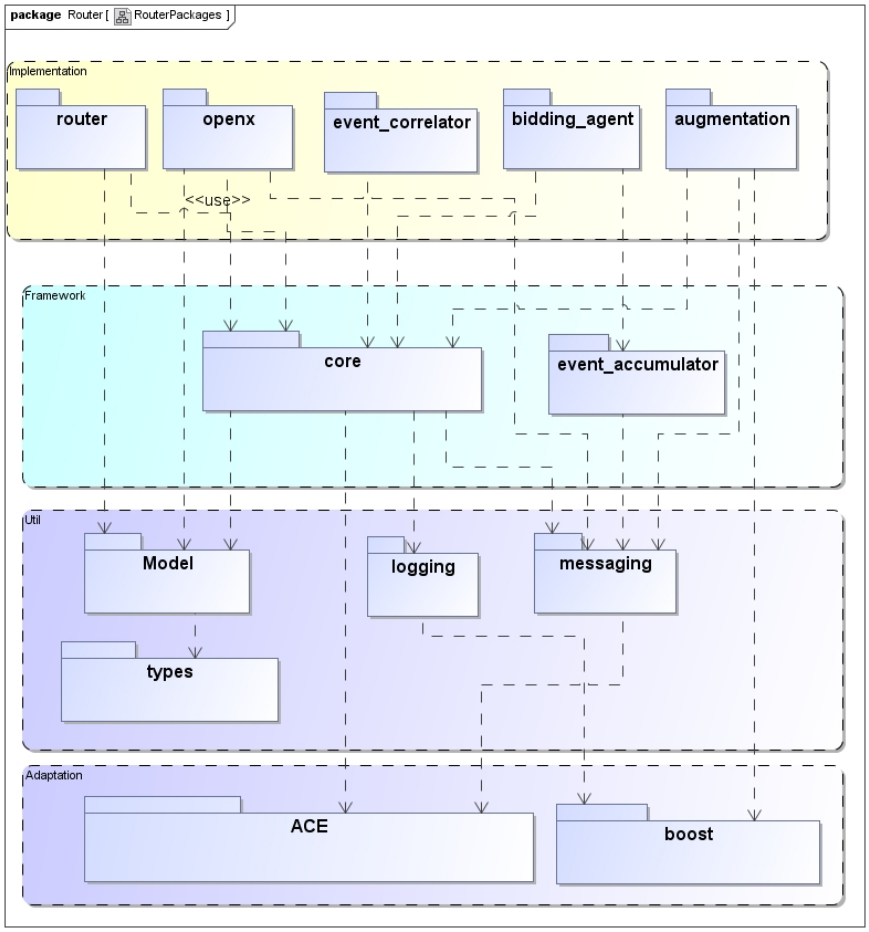


Figure 1 – The Layers of the DSP Router Framework

The DSP Framework is comprised of 4 main layers: Adaptation, Util, Framework and Implementation. All of the layers utilize components within the Adaptation layer, which is composed primarily of the ADAPTIVE Communication Environment (ACE) and Boost. ACE is used for OS and interprocess communication abstraction and adaptation which prevent specific dependencies on the environment that would preclude retargeting the framework for other platforms. Boost provides portable, reusable libraries implementing a wide variety of common data types and algorithms.

The Util layer provides common DSP specific abstractions used to represent DSP data types, communication and

logging mechanisms.

The Framework layer provides the common framework components for implementing RTB and DSP functionality for the Router, Event Correlator and Event Accumulator.

The Implementation layer provides specific implementations of the components using the DSP Framework.

This diagram shows the major components (shown in yellow) in a running instance of the DSP Framework composed of: 1) DSPRouter, 2) ClickHandler, and 3) VoltariBiddingAgent,. Externally implemented processes are shown in grey. The Exchange process represents one of many exchanges that the DSP Router will be interacting with and the generic Device are shown in the diagram as well to illustrate where external events might come from.

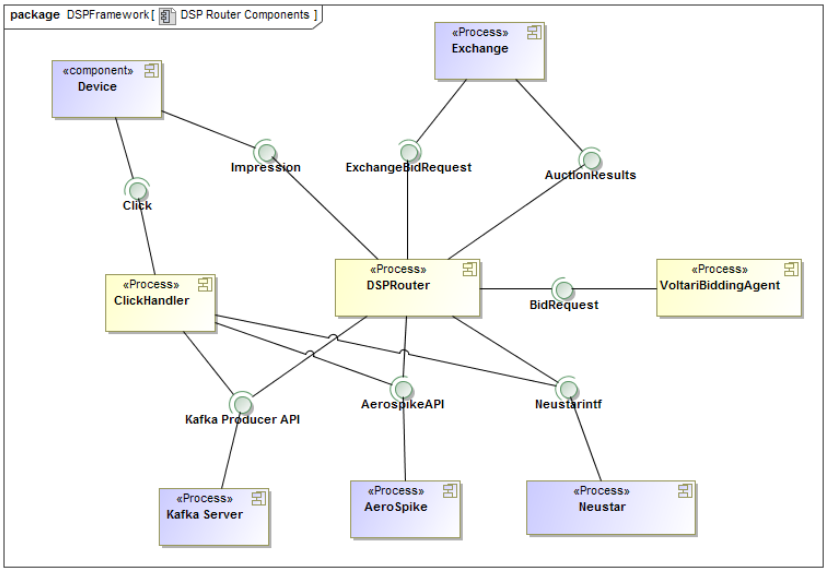
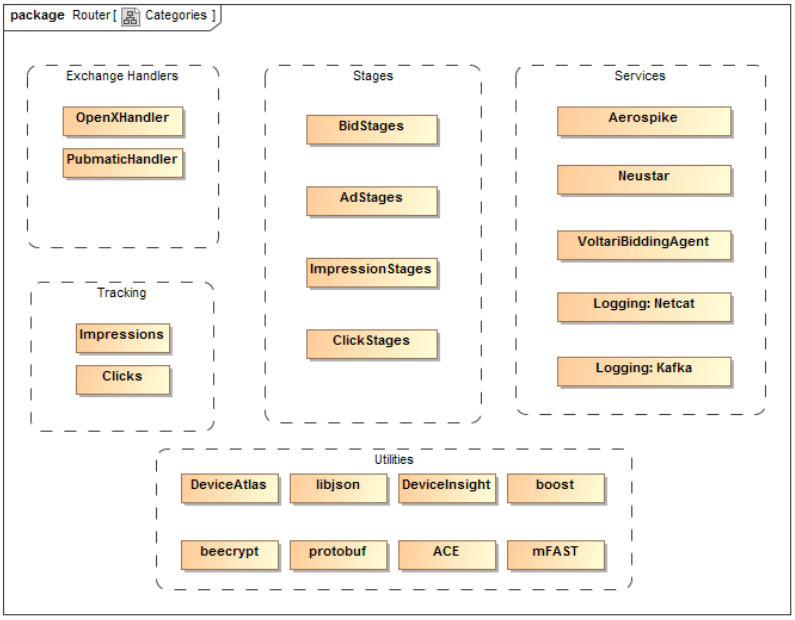


Figure 2. Primary DSP Framework Components

The following diagram shows the primary components within the DSP Router Architecture in 5 different categories (Exchange Handlers, Tracking Handlers, Bid/Impression/Click Stages, Services and Utilities):



In the DSP Router, handlers create stages and stages use utilities and services. A handler is essentially an HTTP server that is setup to process requests asynchronously. It accomplishes this by creating an object to process each request generically called a stages object. A stages object is so named because it is composed of a number of stages organized around the Chain of Responsibility Pattern. Each stage in a stages object uses various utilities and services to accomplish its work. Utilities are local libraries while services represent remote services.

A loader is a dynamic, singleton object that provides internal or external functionality. For example, the Log_Loader configures the logging services (internal) while the OpenX_Loader configures the OpenX exchange handler (external). Some loaders depend on other loaders, e.g., many services depend on the Log_loader. When possible, interfaces were extracted for interal loaders as this reduces coupling and facilitates testing.

The DSP Router is designed around a single-threaded Reactor pattern to avoid blocking on I/O. Where blocking is unavoidable, e.g., Aerospike, an additional thread was added to create an Active Object that inverts the Half-Sync/Half-Async pattern. Other threads exist in libraries that perform background processing such as logging.

Exchange Handlers An exchange handler represents the interface between the DSP Router and an exchange. The common aspects of exchange handlers such as HTTP communication and statistics have been factored out into an exchange handler framework. To use the framework, one must provide the following classes:

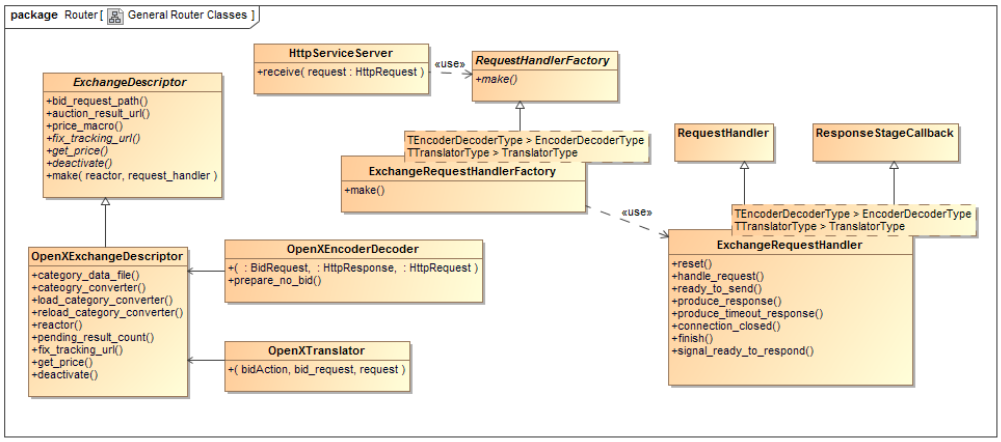
Encoder/Decoder - Class that transforms HTTP requests into an in-memory representation and an in-memory representation into an HTTP response. Often, this is a no-op as an HTTP Request/Response is desirable. For some exchanges, however, the in-memory representation is based on the encoding technology of the exchange, e.g., Google Protobuf.

Translator - Class that translates the in-memory representation to/from a BidAction which is the Router's common representation for all bid requests.

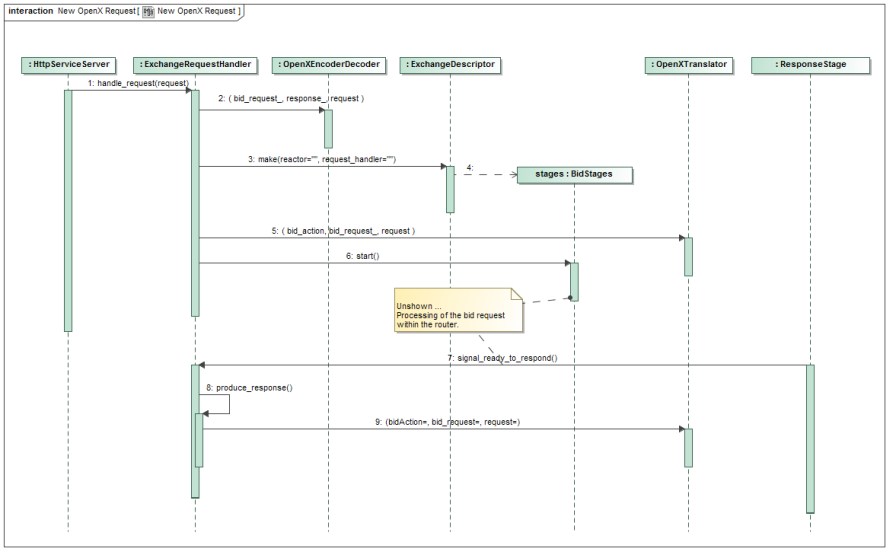
Descriptor - Class that encapsulates exchange-specific details that are need for bid processing.

Loader - A composite object that instatiates the framework with the encoder/decoder, translator, and descriptor.

The following diagram shows the classes which provide the general abstractions needed to implement Router components for a given exchange. In this diagram it shows specific classes used to implement an OpenX instantiation of the framework.



The flow for processing a bid request in the OpenX instantiation is shown below:



Tracking Handlers

Tracking handlers process impression, clicks, and win notifications from devices and exchanges. Tracking handlers are built directly on top of the HTTP services. A tracking handler is typically organized as follows:

Factory - There is a factory that produces a RequestHandler for each HttpRequest.

Converters - The factory is often based on a converts which handle different paths and produce different RequestHandlers.

Stages - A converter makes a Stages object which is a RequestHandler.

Stages

Stages are organized around the Chain of Responsiblity Pattern. A stage may either be synchronous, meaning that it will execute and start the next stage immediately or asynchronous meaning that it will execute and resume after some period of time. If a stage interacts with a service, then it should be an asynchronous stage to avoid blocking behavior. Stages can either be executed or skipped. When a stage is executed, it performs its normal processing task. When a stage is skipped, the stage may assume that a prior stage failed. As such, it may perform its normal processing task or it may skip it if it is not necessary. When a stage fails it should invalidate the entire chain. This avoids useless processing.

BidStages

BidStages are used by public exchange handlers. The following are the invidual stages that comprise a BidStages object in order:

1. RequiredFieldsStage - Checks that the bid request contains fields required for processing.
2. LocationFixupStage - Manipulates location information.
3. ReverseGeocodeStage - Performs a reverse geocode lookup if possible.
4. NeustarStage - Performs a reverse IP lookup.
5. TimezoneStage - Determines the timezone of the device.
6. DeviceStage - Performs device identification using DeviceAtlas.
7. ReadVoltariDeviceIdStage - Reads the VoltariDeviceId for the device.
8. WriteVoltariDeviceIdStage - Writes the VoltariDeviceId for the device.
9. SegmentLookupStage - Looks up the segments for this device.
10. CreativeFilterStage - Determines the set of eligible line items.
11. FrequencyCappingStage - Removes line items on the basis of frequency (device impression rate limit).
12. PacingStage - Removes line items on the basis of packing (delivery goal of line item).
13. CreativeSelectionStage - Selects creatives from each eligible line item.
14. InitialPricingStage - Determines an initial bid price.
15. BiddingAgentStage - Sends the bid request to the bidding agent.
16. ExpansionStage - Processes the response from the bidding agent and generates ad markup.
17. TrackingStage - Inserts the bid request into Aerospike for tracking.
18. ResponseStage - Sends the response to the exchange.
19. LoggingStage - Inserts an entry into the bid log.
20. CleanupStage - Cleans up the BidStages object.

AdStages

AdStages are used for private exchanges and similar to BidStages.

ImpressionStages

The ImpressionStages fetch a bid request from the database, update various fields, commit the bid request back to the database, and insert and entry into the impresison log.

ClickStages

ClickStages retrieve a bid request from the database, update various fields, commit the bid request to the database, and insert an entry into the click log. ClickStages also perform a Neustar reverse IP lookup and may update device

identifiers if AdTruth is enabled. The ClickStages is designed to handle various click scenarios such a redirection vs. pixel-fire and first-click vs. second-click.

Services and Utilities

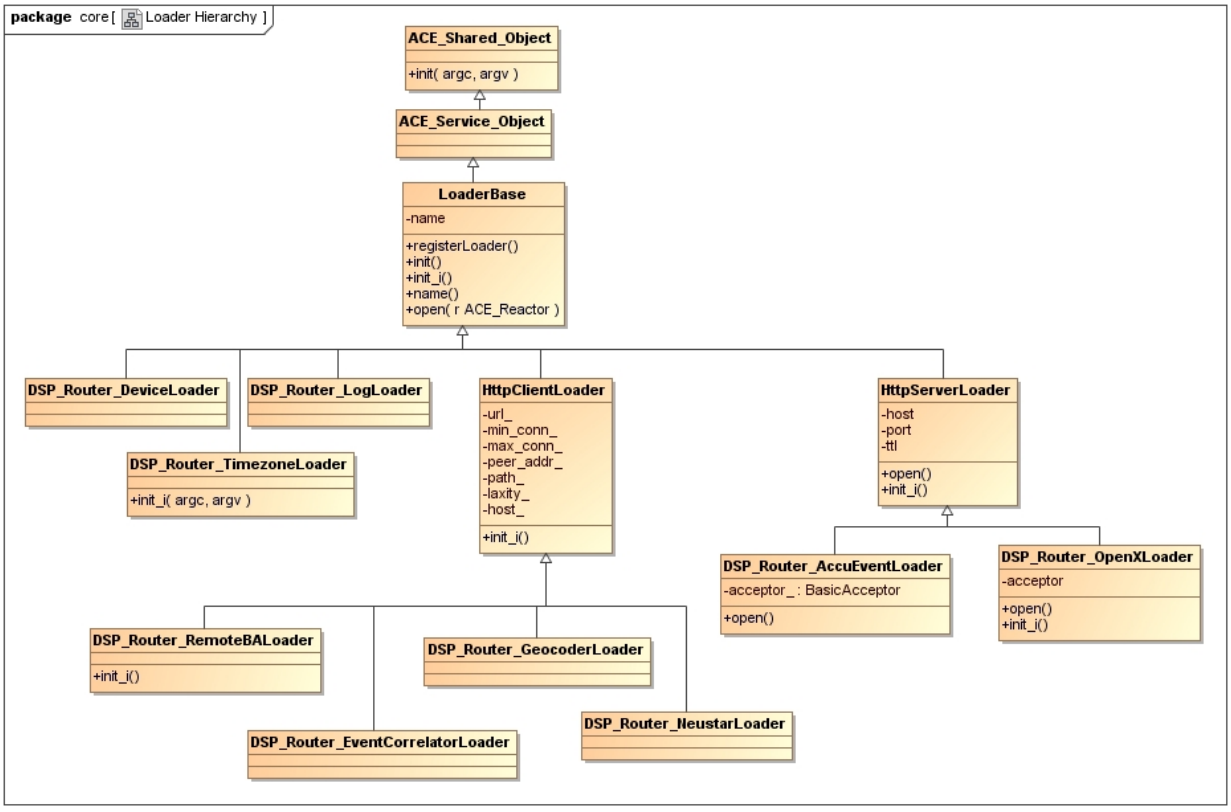
The three main services used by the router are the Neustar Reverse IP service, the Voltari Bidding Agent, and the Aerospike Database. Neustar and the Bidding Agent are web services. The Aerospike database is abstracted via a persistence interface that contains a synchronous part for testing and an asynchronous part for production. (The asynchronous part dispatches to the synchronous part.) The Aerospike implementation is an inversion of the Half-Sync/Half-Async pattern that posts back to the main Reactor thread. The Neustar and Bidding Agent services use the Reactor thread directly.

The other services that the Router uses is various logging services such a Boost Logging and Kafka via rdkafka. These services have their own internal threads that only have one-way interactions with the Reactor thread.

The router uses various utilities and additional utilities can be added in the usual ways.

Configuration

The components of the DSPRouter are configured using the ACE Service Configuration Framework. The following diagram shows the primary components of the DSPRouter configuration which implement the ACE_Service_Object class as a set of “Loaders”.



Utilities

Several utility abstractions are provided in the DSP Framework to assist in developing each of the DSP Framework components.

HttpClients

The following diagram shows request processing bid stage classes. These stages issue requests and wait for responses within the request processing chain. These stages do not block the processing of the DSP router. Instead, they issue a request to some remote service and return control back to the framework to allow it to enable another stage to continue processing when either a response has been received or a timeout has occurred. In this way the DSP router allows multiple bid chains to execute simultaneously.

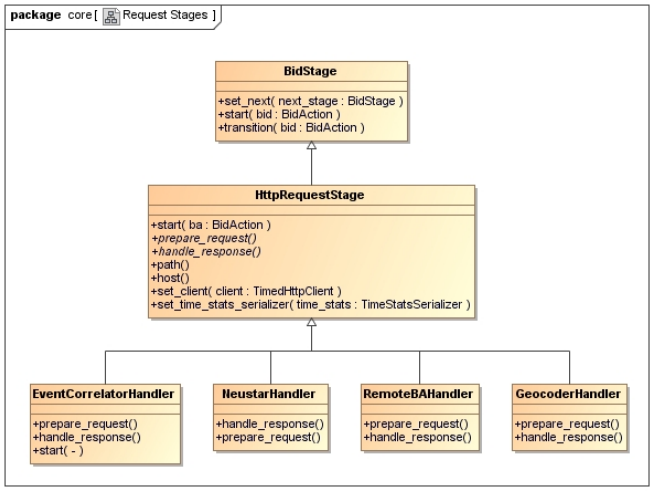


Figure 8. DSP Framework Request Stage (Handlers)

Each request stage can be configured with a set of configuration parameters including the min and max number of connections and the laxity. The laxity is used to calculate the deadline of the stage based on the time to live for the request / response portion of the bid chain. In other words, with a ttl of 120 ms and a stage with a laxity of 20 m, the deadline

Server Ports

The following diagram shows the primary classes for implementing a server object within the DSP Framework. These server objects receive requests from other clients within one of the other DSP Framework processes.

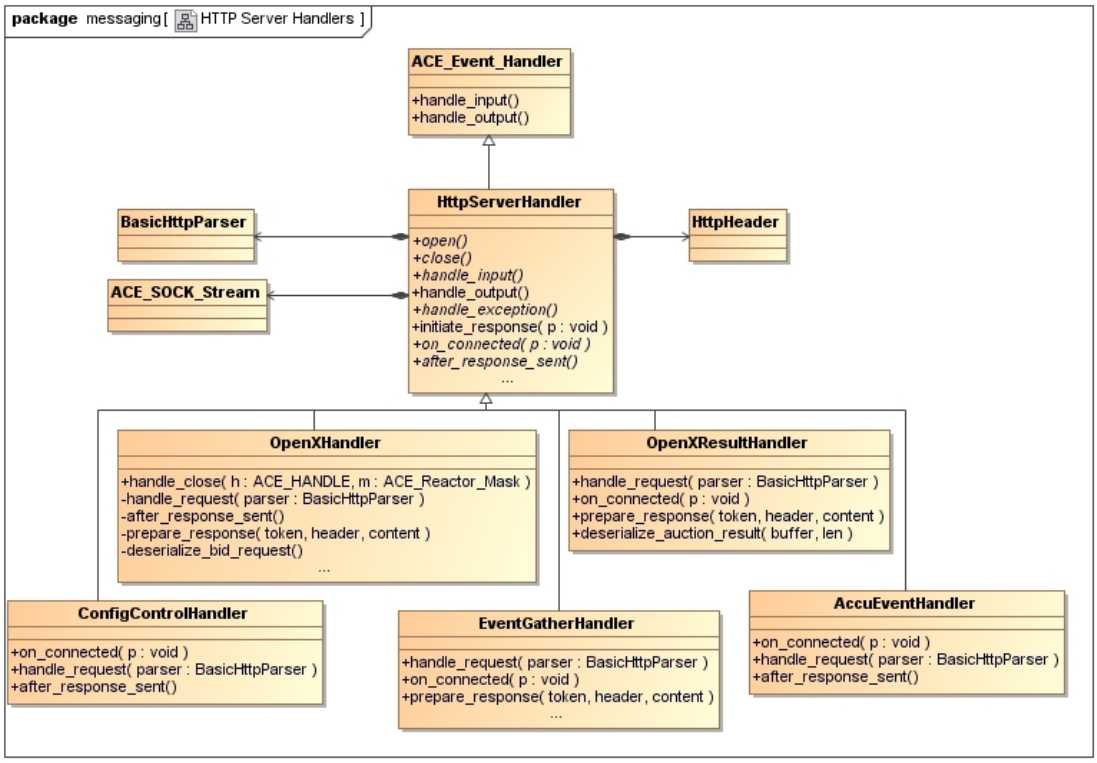


Figure 9. Server Handlers in DSP Framework

HttpServerHandler attributes can be configured using a configuration file at startup or through the administrative console. The following general configurable attributes apply to ServerHandlers: host, port, and ttl. The host and port parameters are associated with the service acceptor host and port respectively. The ttl (time to live) parameter indicates the amount of time to provide the response back to the client.

Logging

The logging facility API is an instance of the boost::log facility available with boost version 1.54. This facility allows for a very flexible and robust logging facility to be configured. It is set up in the DSP software to log the required persistent data to external log aggregators - expected to be flume processes. Additional data sinks can be configured during development and test as well as in place to gather additional diagnostic information during production.

The logging service is included into the applications by using the service configurator. This allows the logging service to be configured and loaded independently of the remainder of the application. It also allows a dedicated command line to be passed to the service.

The logging API allows the application to insert log messages and have them emitted to one of several configurable destinations using an insertion operator like below:

```
DSP_LOG_BID_EVENT << bid_event.to_string();
```

The logging service is configured via file. Currently this is an 'ini' format file with key/value pair in sections. Each "Sinks" subsection defines a different backend. The existing event logging back ends use Channel Loggers, which rely on the "Channel" attribute value to determine which log records to emit.

DSP Logging allows for time-based rotation which can be set up with one of the two parameters: RotationInterval or RotationTimePoint.

The standard boost log service back-ends are available and can be used to capture log records. Some back-end types can be configured directly from a configuration file, and includes:

```
Console - write log records to the current console device.

TextFile - write log records to text files, possibly performing storage
management (rotate and remove) operations as well.

Syslog - write log records to either a local or remote syslog facility.
```

Backends of these types can only be created and configured programmatically:

```
TextStream - write log records to a text stream.

TextMultiFile - write log records to multiple text files simultaneously.
```

In addition to these standard back-ends, a local implementation for connecting to a TCP port and sending log records over that connection is included in the application.

The logging component in the DSP Router can be configured using an file at startup. The file format is described here: https://extranet.ociweb.com/redmine/projects/2013-0053/wiki/Logging#Configuration-File-Format

DSP Router

This diagram shows the notional internal functional elements of the DSP Router (middle of the diagram above), showing which external interfaces they interact with. These correspond to the ports on the DSP router in the diagram above. These components were defined to address appropriate functionality described in the use cases.

Auction Processing (Chain of Responsibility Pattern)

The primary pattern supporting the DSP Framework is Chain of Responsibility [GOF Design Patterns]. In the DSP Framework a primary use case is Handling a Public Bid Request. Requests are processed using a chain composed of a sequence of stages that process the request, send it to a bidding agent, reply to the exchange and wait for an auction result. Each HttpRequestStage in the chain follows a pattern shown in the following sequence diagram.

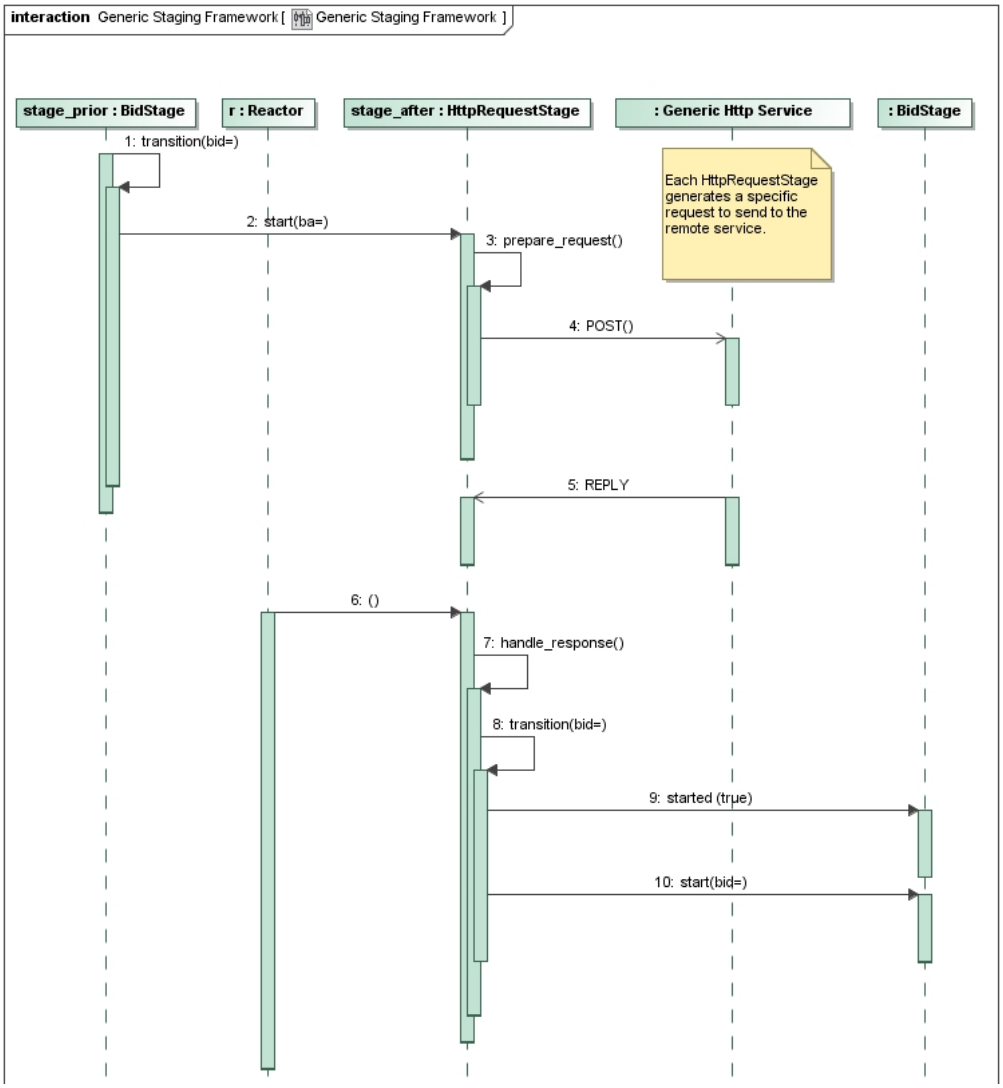


Figure 3. General Bid Stage Sequence Diagram

The Bid Stages for OpenX are shown in the following figure.

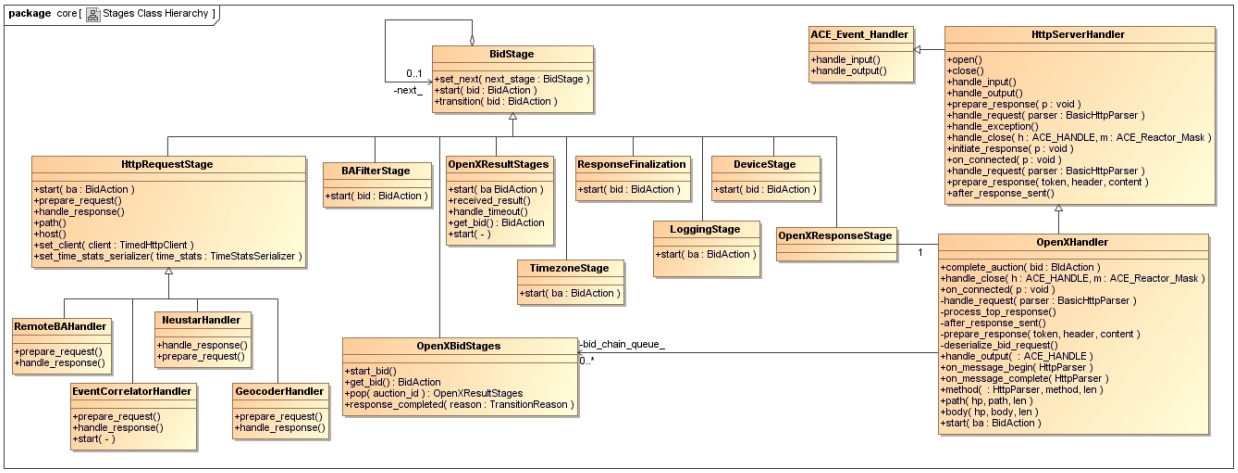


Figure 4. Bid Stages Class Hierarchy

The following collaboration diagram outlines the primary flow for processing a Bid Request from OpenX. The chain is created by the OpenXHandler, which constructs the chain and starts the processing (via the start_bid() operation). The processing includes using Geocoder, Neustar, Timezone and Device augmentation stages followed by Bid Agent Filering and Remote Bidding Agent processing. This is followed by stages for receiving the response from OpenX , logging the request and response and finally waiting for the OpenX result and submitting the Win Notification to the Event Correlator and Logging the result. Note that some of the stages have a stereotype <<HttpRequestStage>> which indicates that the stage include sending a request to a remote service and waiting for the response. The processing of a chain is suspended waiting for the response or a timeout, which allows other active chains to process other requests. In this manner the asynchronous architecture of the DSP router interleaves processing of requests in a single thread.

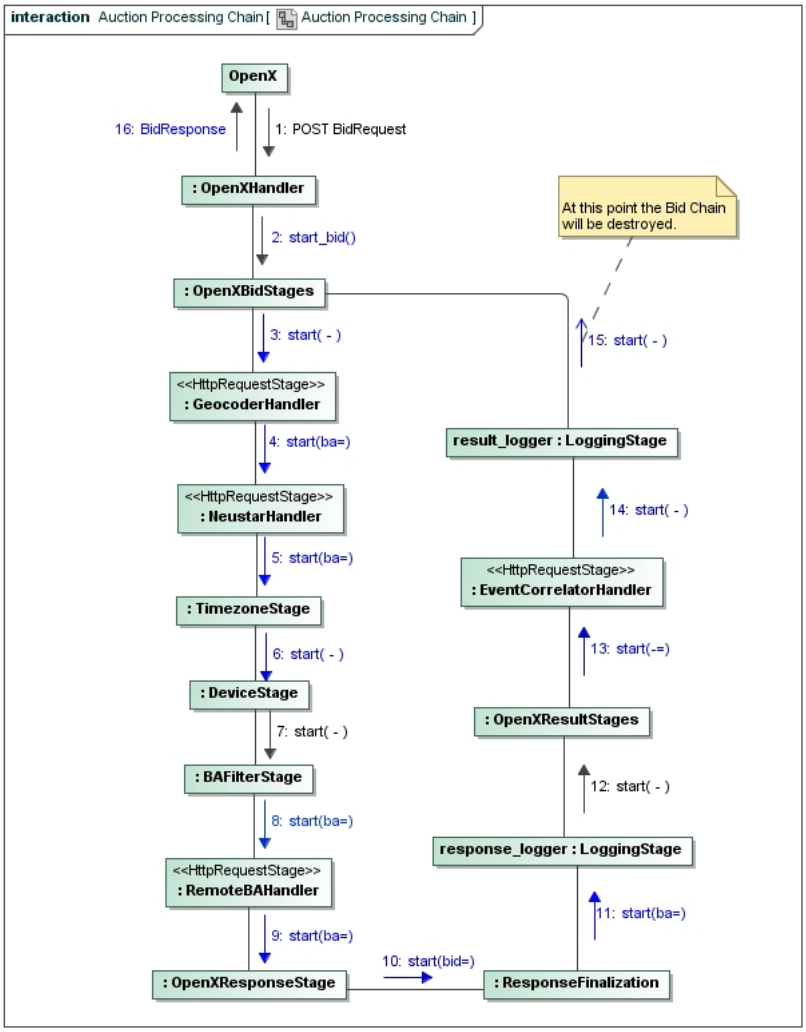


Figure 5. OpenX Router Bid Stages

Exchange Connection Management

The DSPRouter accepts connections from an exchange on a dedicated endpoint. When a connection is established a Handler object is created that will process the BidRequests. For OpenX, the BidRequest comes in the form of an HTTP POST which contains a Protobuf encoded structure that contains the BidRequest details.

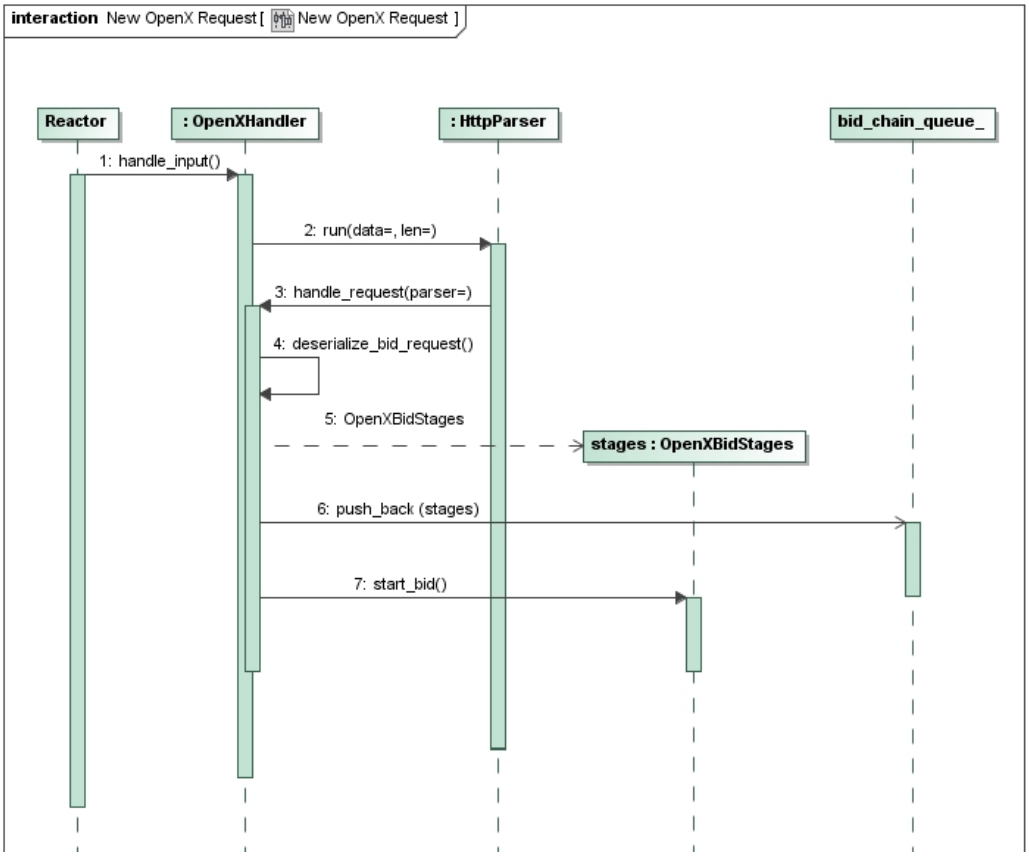


Figure 6. New OpenX Request

Upon receipt of a new request, the handler (OpenXHandler – for the OpenX Marketplace implementation) will parse the request and construct a new chain, which is instantiated by the OpenXBidStages class (as shown in step 5 above). To ensure that we respond to bids on the connection in the order they were received, we push the top stage onto a queue. We then initiate the chain by invoking the start_bid() method on the the OpenXBidStage object that was created.

From another perspective, the flow is shown in a collaboration diagram below. Note that only a few of the stages are shown in the diagram, with a note representing the intervening stages. The OpenXHandler object constructs the top bid stage and the same OpenXHandler object then is utilized by the OpenXResponse stage to submit the response back to the OpenX exchange on the same connection.

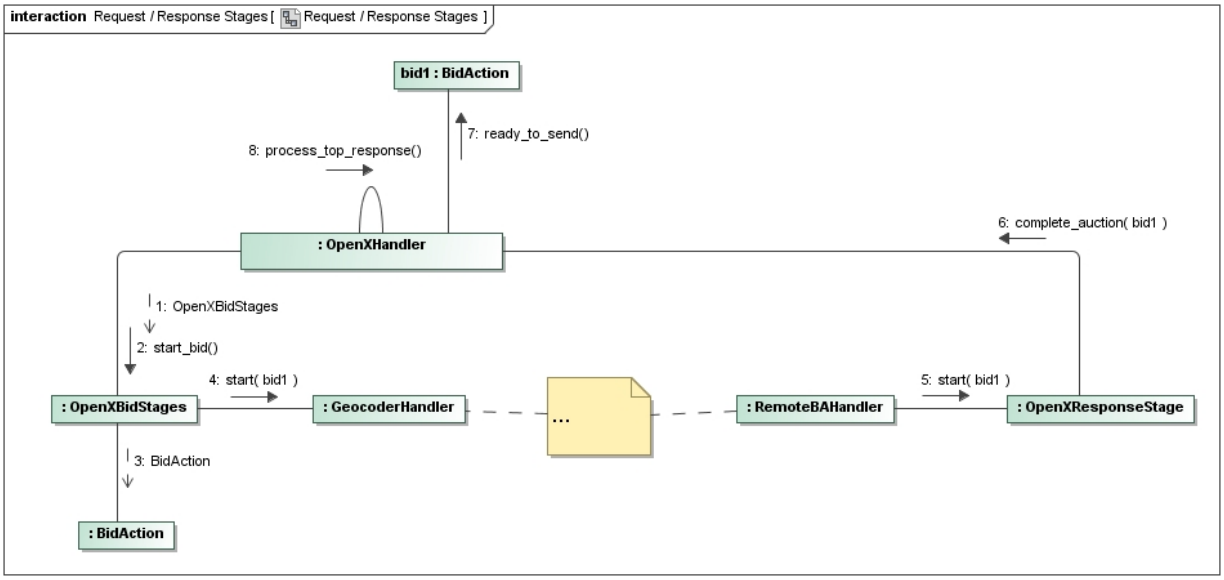


Figure 7. Request / Response Process Flow

Augmentation

The DSP Framework currently supports augmentation stages for Reverse Geocode, Neustar, Device Property Management, and Timezone management. The Reverse Geocoder and Neustar augmenters are implemented as Web Services and require interprocess communication to retrieve the augmentation data, while the others are implemented as in-process functions. These augmentation stages can be reused in other instantiations of the DSP Framework – e.g., for other exchanges.