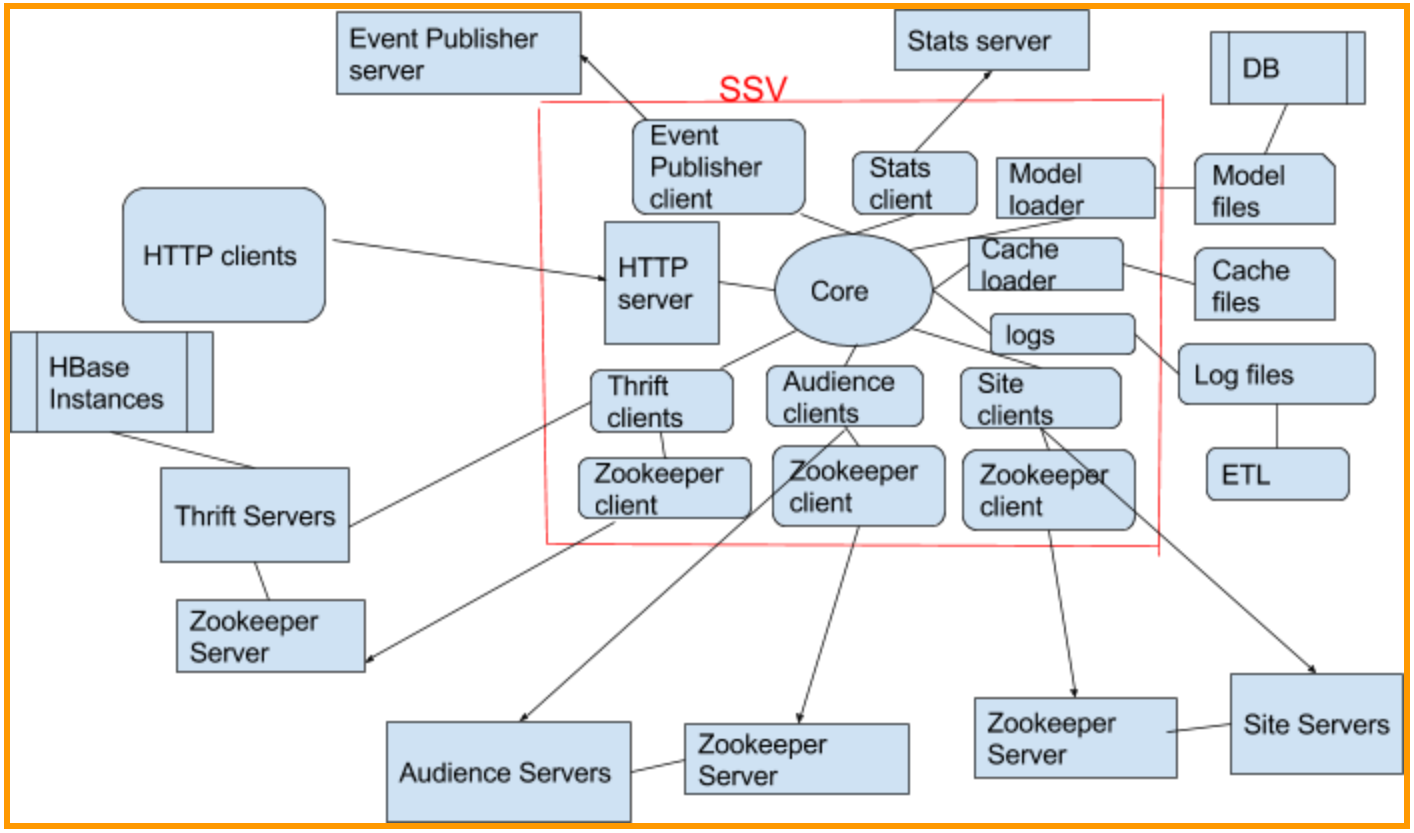
SSV - communications.

Communication Diagram.



HTTP server.

HTTP server creates configurable number of request processing threads (in current configuration the thread pool has 200 threads) and listens for http requests from clients. When a client connects and sends request - then server: 1. distributes request to processing thread , 2. parses request , 3 calls Core library, Core library produces response, 4. HTTP server encodes response into http message, 6. sends the message back to the client. All processing (from receiving request to sending response) in one thread.

Notes. The threading model is going to be changed in the next version of the HTTP server. It is enforced that http client always receives response even if the request is bad. Current HTTP servers in production configured for non-secure http 1.1, but also could accept SPDY and use SSL security. Next version of the HTTP server also supports http 2.
HTTP server implemented as a static library libssv_web_lib.a , source code in ssv_web_lib.

Zookeeper client.

Zookeeper client does not have associated business functionality and used by thrift, audience and site clients. Zookeeper client connects to zookeeper server and reads the list of associated servers (e.g. zookeeper client associated with thrift clients reads list of the thrift servers.) Only then clients connect to the thrift servers from the list. Choice of servers happened randomly (which means evenly). Zookeeper client runs in its own thread and updates the server list at run time if the list has been changed.

Notes. Zookeeper client implemented as part of a static library libzoo_load_balancer, source code in zoo_load_balancer/zoo_adapter.

Thrift clients

Thrift clients library has several major components: pool of connected and registred thrift clients (“main” pool), pool of recovering thrift clients (“recovery” pool), zoo adapter, load balancer.
On startup zoo adapter creates list of available thrift servers (~ 10-20 servers). Thrift clients from the “main” pool (in current configuration ~ 200 clients) distributed randomly over the thrift servers. Thrift clients connected and registered with the thrift servers.
If at runtime request from request processing thread, needs data from Hbase (or needs to send data to Hbase) then it pool a client from the “main” pool. Communicates with Hbase (using thrift server). Push the client back to the pool.
If the corresponding thrift server went down or communication between thrift server and Hbase is “bad” then it returns a

default http message to the http client and trying to recover asynchronously in “recovery” pool. Then it push client back from “recovery” pool to the “main” pool.
Notes. Dynamic (runtime) load balancing and recovery looks relatively complicated, but mandatory in the current system. It was proven by monitoring in production that without this mechanism the error rate is ~ 10 - 20%. Such rate considered to be not acceptable for DMP. Error rate with dynamic load rebalancing and recovery is 1-2%.
Thrift clients library implemented as a static library (libhbase_thrift_client.a), source code in hbase_thrift_clien.

Audience clients.

Audience clients component has functionality similar to thrift clients. The main differences: protocal - TCP-ZMQ with protobuf encoding, no registration with the server, synchronous recovery . Major components: pool of connected audience clients, zoo adapter, load balancer.
On startup zoo adapter creates list of available audience servers (~ 10-20 servers). Audience clients from the pool (in current configuration ~ 200 clients) distributed randomly over the audience servers. Audience clients connected with the audience servers.
If at runtime request from request processing thread, needs data from audience server then it pools a client from the pool. Communicates with audience server. Push the client back to the pool.
If the corresponding audience server went down then it returns a default http message to the http client and trying to recover synchronously.

Site clients.

Site clients component has functionality identical (in terms of communication and recovery mexanizm) with audience clients.The only difference is that it connects with different list of site servers and requests different data.

Cache loader.

Cache loader loads a cache from cache files on startup and on corresponding http requests. In production those requests generated by scripts.

Model loader.

Model loader loads cache from model files, on startup and on corresponding http requests. In production those requests generated by scripts.

Stats client

Stats client reports stats to the stats server. Used for monitoring.

Event Publisher client.

Event Publisher client reports some events to Event Publisher server.

Problems and ideas.

Problems.

1. We have too many runtime communication mechanisms and protocols(http, zookeeper, thrift, ZMQ, file loading and file generation).
2. Some communications are too slow and not reliable enough. Thrift clients, audience clients, site clients and (may be) model loader have this problem. Error rate ~ 20% is not acceptable and we should avoid complicated rebalancing and recovery on the clients (SSV) site.

We should try:

1. Minimize number of communication mechanisms and protocols.
2. Make communication faster.
3. Make communication more reliable.
4. Separating communication and business logic seems to be good idea.

Ideas

1. As the first step we can move zookeeper clients, thrift clients, audience clients and site clients from SSV to a separate Data server and make it faster. (see “Suggested design. First step” below). Communications between the Data server

and SSV could be done with a fast binary protocol (e.g. protobuf over TCP). One Data server per data-center. This step seems to be not too complicated. Caching data in this Data server seems to be effective optimization for the DMP (needs to be verified). Appropriate cache implementation still needs to be defined. Igor can do this step.
2. We also can try to move more data from Hbase, audience and site servers to the Pixel server and send the data to SSV with http requests. (Move all data to Pixel server could be difficult, because Pixel server does not know SSV internals).
3. Since Pixel server parses http, we can replace slow http protocol (between Pixel server and SSV) with faster binary protocol (e.g. protobuf over TCP).
4. We can load models directly from DB.

Suggested design. First step