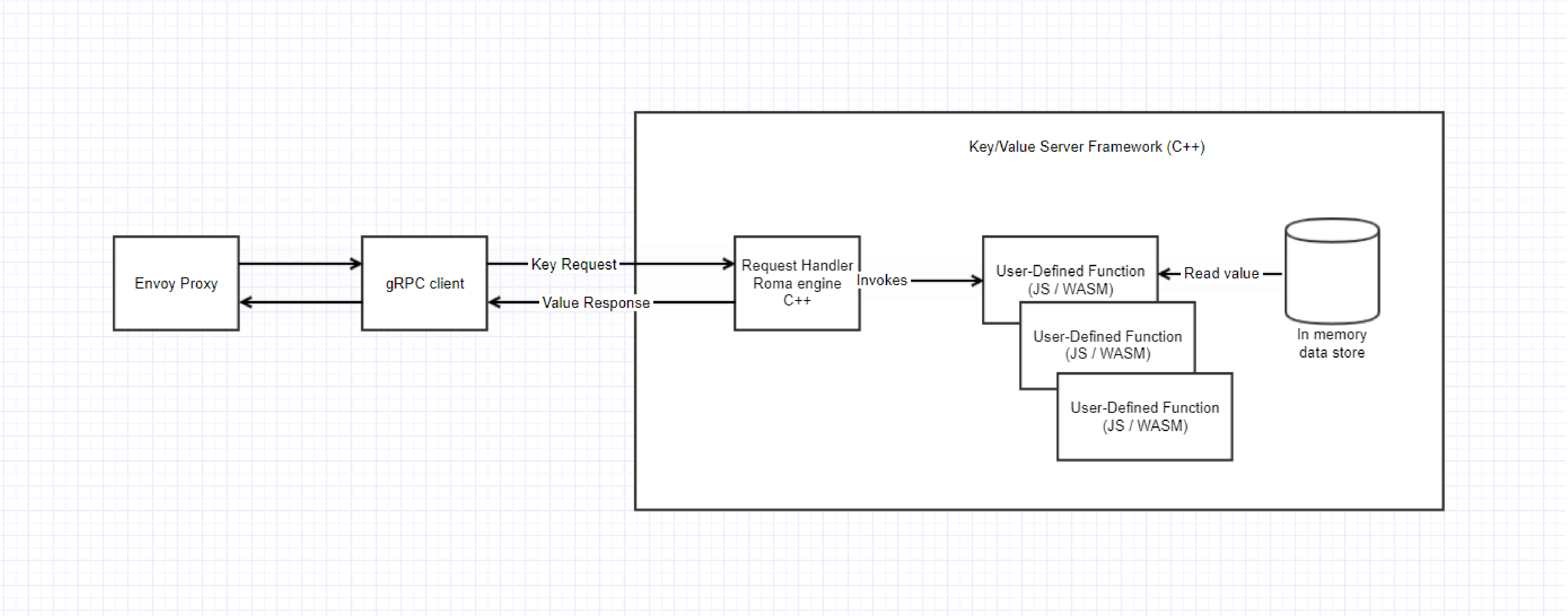
# Key value server / ROMA benchmarks

Fabian Höring

f.horing@criteo.com

# Non TEE server setup on our own infra



## Server setup

Deploy locally compiled key-value server on 1 instance with 16 cores & 16 GB of memory with a very basic implementation

- No batching
- it reads the keys from the request
- it queries the in-memory datastore to get the values of those keys
- it replies with those values
- C# asp.net vs Google key/value service with same input & output

# Web load test with gatling

- Traffic dump with calls to the key/value server (keys 1 to 1000)
- QPS 500-100000
- Assert  results for server payload

# Results ASP.NET vs KV server

| Service | mean response time | 90 percentile | 99 percentile | max QPS | max Memory |
|---|---|---|---|---|---|
| c# fledge bidding | 3ms | 5ms | 9ms | 45k | 2.5 GB |
| c# fledge bidding (at 10k QPS) | 1ms | 2ms | 2ms | 10k | 2.5 GB |
| key value service | 6ms | 8ms | 9ms | 5k | 1 GB |

# Conclusion ASP.NET vs KV server

- Google implementation can handle 1000s of queries per second with ms second latency
- Asp.net can handle 10 times more queries (45k QPS vs 5k QPS) with better mean response times (3ms vs 6ms)

# WASM with c#/c++

- Emscripten is an LLVM/Clang-based compiler that compiles C and C++ source code to WebAssembly

```
async function getModule() {
    var Module = {
        instantiateWasm: function (imports, successCallback) {
            var module = new WebAssembly.Module(wasm_array);
            var instance = new WebAssembly.Instance(module, imports);
            Module.testWasmInstantiationSucceeded = 1;
            successCallback(instance);
            return instance.exports;
        },
    };
    return await wasmModule(Module);
}
```

# WASM with dotnet (dotnet 9 preview)

```
dotnet workload install wasi-experimental
```

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <RuntimeIdentifier>wasi-wasm</RuntimeIdentifier>
    <OutputType>Exe</OutputType>
    <PublishTrimmed>true</PublishTrimmed>
    <WasmSingleFileBundle>true</WasmSingleFileBundle>
    <EventSourceSupport>false</EventSourceSupport>
    <UseSystemResourceKeys>true</UseSystemResourceKeys>
    <EnableUnsafeUTF7Encoding>false</EnableUnsafeUTF7Encoding>
    <HttpActivityPropagationSupport>false</HttpActivityPropagationSupport>
    <DebuggerSupport>false</DebuggerSupport>
  </PropertyGroup>
</Project>
```

# WASM Results

| Service | mean response time | 90 percentile | 99 percentile | max QPS | max Memory | Max CPU |
|---|---|---|---|---|---|---|
| c++=> WASM | 9ms | 12ms | 16ms | 1.2k | 1 GB | 15.4 |
| c# => WASM | 3 sec | 3 sec | 3 sec | 1 (without the K) | 1 GB | |

# Conclusion WASM

- c++ WASM can handle 1k QPS (file size 100 KB), c# WASM can handle 1 QPS (file size 30 MB)
- WASM doesn't seem like a real alternative, it currently needs to compile the file at each request

# Efficient In memory caching

- Currently all data lookups from bidding script to KV state need to be serialized/de-serialized
- For real time bidding server must reply within 50ms
- **Optimizations here actually matter** (where we might not care much elsewhere)
- Network calls and IPC should be reduced or batched
- Over 30 caches in our Prod systems on PA POC

# Efficient In memory caching, examples

- Efficient filtering, sending back all campaigns by country & all campaigns by domain and doing the intersection in the bidding layer
- ML Inference Sidecar, Inference side will require IPC, overhead will be significative, times 5 with internal ONX benchmarks

# One option – inline datastructures

## Inline large data structures for caching #62

Open fhoering opened this issue on Jun 26 · 5 comments

**fhoering** commented on Jun 26 • edited

Caching is an important mechanism for improving performance. I tried to inline a large datastructure to be able to reuse it in a readonly mode without querying the KV storage again. But using this the performance is getting very bad.

Can you have a look where the downlift is coming from ? Can it come from the large file sizes or from reinitializing the array all the time ?

```
const space = 10000000
const w = [0.004996137771039466,0.7571896910549433,0.7937806398622418,0.9647932057841067,0.00228668

function HandleRequest(executionMetadata, metadataKeys, signal) {
        let result = 0;
        for (let i = 0; i < 100; i++) {
            const index = Math.floor(Math.random() * space);
            result += w[index];
        }
      return result
    }
};
```

# Inlined datastructures

- JS engine doesn't handle inlined floats & structures in an efficient way
- Tested proposed option inlining ArrayBuffer with 1 mio elements

```
const buffer = new Uint8Array([1, 2, 3, 4]);
const floatArray = new Float32Array(buffer.buffer);

function HandleRequest(executionMetadata, ...udf_arguments) {
  for (let argument of udf_arguments) {
    let result = {};
    let key = argument.data;
    result[key] = floatArray[0];
    return result;
  }
};
```

# Pure JS baseline

- getValues hook to access KV state
- Tested JS performance without getting state

```
function HandleRequest(executionMetadata, ...udf_arguments) {
  for (let argument of udf_arguments) {
    let key = argument.data;
    const getValuesResult = JSON.parse(getValues(key));
    return getValuesResult;
  }
};
```

# In memory caching

| Service | mean response time | 90 percentile | 99 percentile | max QPS | max Memory | Max CPU |
|---|---|---|---|---|---|---|
| key value service | 6ms | 8ms | 9ms | 5k | 1 GB | 15 |
| kv service, inlined array | 7ms | 10ms | 12ms | 4.2k QPS | 1 GB | 15 |
| kv service, no getvalues hook, (pure JS baseline) | 7ms | 10ms | 12ms | 6k QPS | 1 GB | 15 |

# Conclusion in memory caching

- Didn't even try to deserialize a large object yet from getValues
- Pure JS baseline vs KV service => **+20%**
- We have 30 caches in production
- Explore Shared memory or Long running processes

# Server Infra cost might quickly get out of control

- JS ROMA vs Native c# => **x5**
- No shared memory => **+50%** (30 caches in prod system)
- ML inference side car => **x5** on ML inference (internal ONNX benchmark)
- TEE and encrypted networking => **+20%** (usual symmetric encryption overhead)

# Future work

More work is needed to mitigate potential infra cost increase

- Shared memory or Inlining data structures
- New ROMA BringYourOwn binary execution engine to allow custom languages beyond JS & WASM

Common method of benchmarking, improvements should be trackable easily, web load tests