GTS

General Trading System Demo Architecture Draft

Communication Architecture

Legend:

CIL - Communication Interface Layer

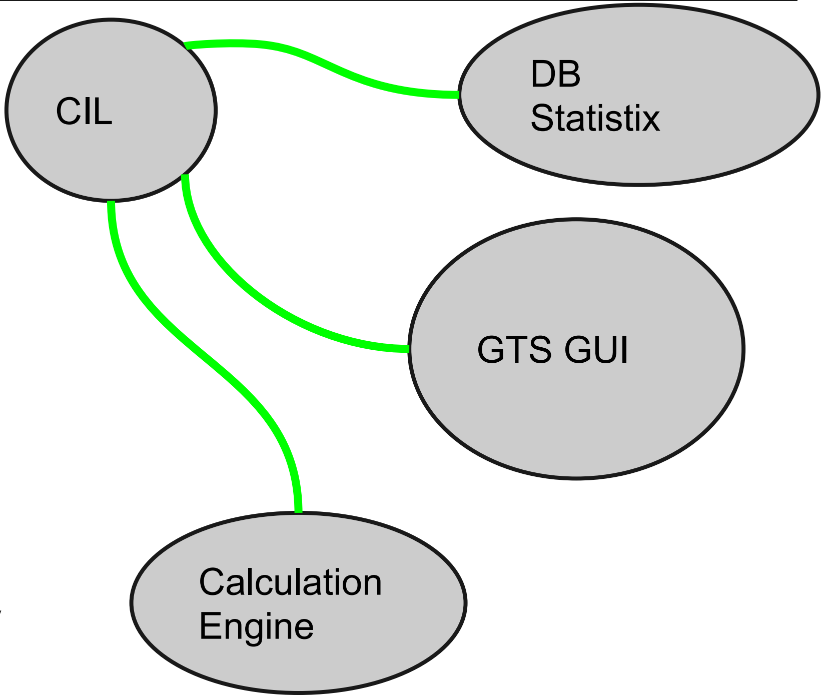
Calculation Engine - various evaluations on historical data, prepares risk factors, risk factor values, prepares samples, margins, backtesting, stress-testing …

Calculation Engine has no DB -

all data is read/stored on DB external aka **Statistix**

Communication is provided by ZeroMQ broker on CIL

Data is being transported using messages formatted by Google Protocol Buffers.



ZeroMQ as the messaging layer

The transport layer can be made via INPROC, IPC, MULTICAST, TCP. I’ve chosen TCP which provides unicast transport using TCP, the reason for the decision was because there is no need to cross the machine border. The multicast is out of choice because it behaves difficult when scaling up (while crossing many switches over network).

The Infrastructure pattern - QUEUE, FORWARDER and STREAMER.

The Message Pattern is based on Request/Reply bidirectional, load balanced and state based or publish/subscribe to publish to multiple recipients at once, upstream/downstream - data arranged in a pipeline.

Data model

General entities - via Espresso - automatic generation.

Manually modelled entities - compile via protoc

Development Environment

Actually this point is not implemented yet, I would suggest to use CMakeLists.txt for automatic build and linkage. The CmakeLists.txt would define what database model will be used to compile with the module.

The advantage of cmake usage is possibility to fetch all project build to Jenkins automation environment.

Data access to Statistix 1/2

Write access

- insert, update and delete store requests
- always the timestamp should be used
- best performance would be reached if store requests are sorted per entity and type I/U/D, otherwise there is awaited very poor responsivity
- Data cache (as unordered boost::unordered map), not yet implemented

Data acces to Statistix 2/2

- Read access
- for Open Snapshot (timestamp)
- joins won’t be supported, instead is recommended to use views
- possbility to use BOQL interface, i.e. restricted SQL-like and XML formatted queries
- Roundtrip 0.3 seconds for avoiding many repeated queries, preferred to use data caches loaded at once in one query, use of parallelization
- results returned as a recordset (Bulk Query) or as single Entities (Cursor Quer) - not implemented in demo
- Db - interface - abstract classes for data warehouse data access
- it should be implemented in more languages because of joining more modules (e.g. GUI, reporting etc.), it is recommended Java, Python, C++,...

Query builder for database

there is me not known comfortable query builder database library in c++, probably this should be made in own overheads. I would use placeholders, such as

...Where(Attr(“UUID”) ==LongPlaceholder(“Uuid_) && BooleanExpressionPlaceholder(“filterCondition, true))...

```
static DBCommon::EnregisterQueryBuilder  qbRiskFactor(“gts::pb_riskFactor”,QueryBuilder
(“RiskFactor”).Select(“date”)
                                                                        .Select(“...”)
                                                                        .Where(Attr(“.. etc
                                                                           ).OrderBy(Attr(“date”), DESC));
```

GUI - technical for operators

in terms of GUI there is no limitation on technology, this can be relied on Java, or perl (strong WebGUI languages).

technical GUI consists of controlling workflow of the processes, smoke test, integration test, installation test, sanity checks etc.

There should be connection between job scheduling and this GUI.

Job scheduling

Job scheduling (batch scheduling) - not implemented in demo.

For automatic reporting (html, pdf, xml, …) and sending over email, or making the reports shared, for free it can be used CRON, it is presumed all backed will be running on NIX platform.

Policy based upon request (e.g. failure scenario, crash etc.) - not implemented in demo