Überlegungen zu node_hierarchy - Version 4.0 (Codename: cybär)

PRE-DRAFT

Motivation

Für viele Entscheidungen und Abläufe benötigen wir Daten über unser Netz. Diese haben wir früher mittels *A.L.F.R.E.D.* erfasst, heute erfassen wir sie hingegen mittels *repsondd*. Wir sammeln diese Daten eigentlich nur im Kontext der Kartendarstellung. Dafür nutzen wir derzeit den *hopglass-server*. Jedoch haben wir eine Vielzahl an Anwendungen, die sich diese Daten nehmen und daraus andere Daten ableiten. Dazu gehören unter Anderem:

- *node-stats*: *Brücke* zwischen *hopglass-server* und *Graphite* / Temporale Erfassung von quantitativen Knotendaten
- *node_hierarchy*: Tool zur Optimierung der Firmware-Verteilung und Berücksichtigung von Abhängigkeiten (sowie ein paar Statistiken)
- *geo_info*: Knotenstatistiken nach geographischen Relationen
- *node_stats.py*: Script zur Ermittlung historischer Nutzerentwicklung
- ...

Problem

Bei der Vielzahl an Quellen und Anwendungen werden häufig viele gleiche (oder zumindest sehr ähnliche) Programmteile für jede Anwendung separat entwickelt. Bei Änderungen an den Daten / Datenstrukturen müssen in allen Programmen, teils sehr Umfangreiche, Änderungen vorgenommen werden. Obwohl der Inhalt der Daten meist gleich bleibt werden manchmal sogar ganze Programmteile obsolet bzw. erst erforderlich. Die Wartung dieser Anwendungen ist zum totalen Horror geworden.

Hopglass Server

Der h*opglass-server* (geschrieben in *nodejs*) verspricht prinzipiell eine Abhilfe in dem es (mehr oder weniger) modularisiert ist. Es gibt zwei Typen von Modulen: *receiver* (Datenakquisition, respondd, ...) und provider (Datenlieferanten, *hopglass*, *meshviewer*,...). Leider, dazu genügt ein flüchtiger Blick in den Quelltext, werden hier auch sehr viele Operationen redundant, und somit sehr wahrscheinlich auch inkonsistent, in den einzelnen Modulen gepflegt, obwohl sie auch generisch an **einer** Stelle im Code gelöst werden könnten.

Außerdem ist die Performance der Anwendung eher mäßig. Das liegt zu einen daran, dass es nicht nur redundante Codeblöcke gibt sondern, dass viele dieser Blöcke auch, unnötigerweise, redundant ausgeführt werden. Dutzendfache, redundante, Ausführung auf Daten von mehreren Tausend Knoten und mindestens so vielen Links lässt das Ganze eher im Schneckentempo ablaufen. Außerdem scheint nodejs und insbesondere die Auswahl der verwendeten Frameworks sein Übriges zur eher mäßigen Performance beizutragen.

Abschließend lässt sich noch anmerken, dass man beim Verwenden der h*opglass-server* Struktur an die Sprache *nodejs* gebunden ist.

Bisherige Datenstrukturen

Bisher werden die Daten bei der Erfassung in drei unterschiedlichen Typen geliefert:

- *nodeinfo*
- *statistics*
- *neighbours*

Beim Abruf kann man entweder diese Struktur erhalten (vgl. *raw.json* des *hopglass-server*) oder die Strukturen für *hopglass* bzw. *meshviewer* abrufen. Diese sind unterteilt in:

- *nodes*
- *graph*

Leider befinden sich bei Ersterem die Daten nicht immer dort, wo man sie erwarten würde. Bei Letzterem ist der Zugriff auf die Daten (zumindest bei *graph*) unnötig komplex (vermutlich dem Anstreben einer möglichst geringen Dateigröße geschuldet).

Anforderungen

Daraus lässt sich ableiten, dass die Verwaltung dieser Daten

- unabhängig von Programmiersprache,
- flexibel und erweiterbar,
- individuell abfragbar,
- einigermaßen performant,
- sowie den Strukturen des Netzes entsprechend

erfolgen soll.

Aus diesen Punkten lässt sich folgern, dass die Umsetzung in einem relationalen Datenbankmodell als geeignet erscheint.

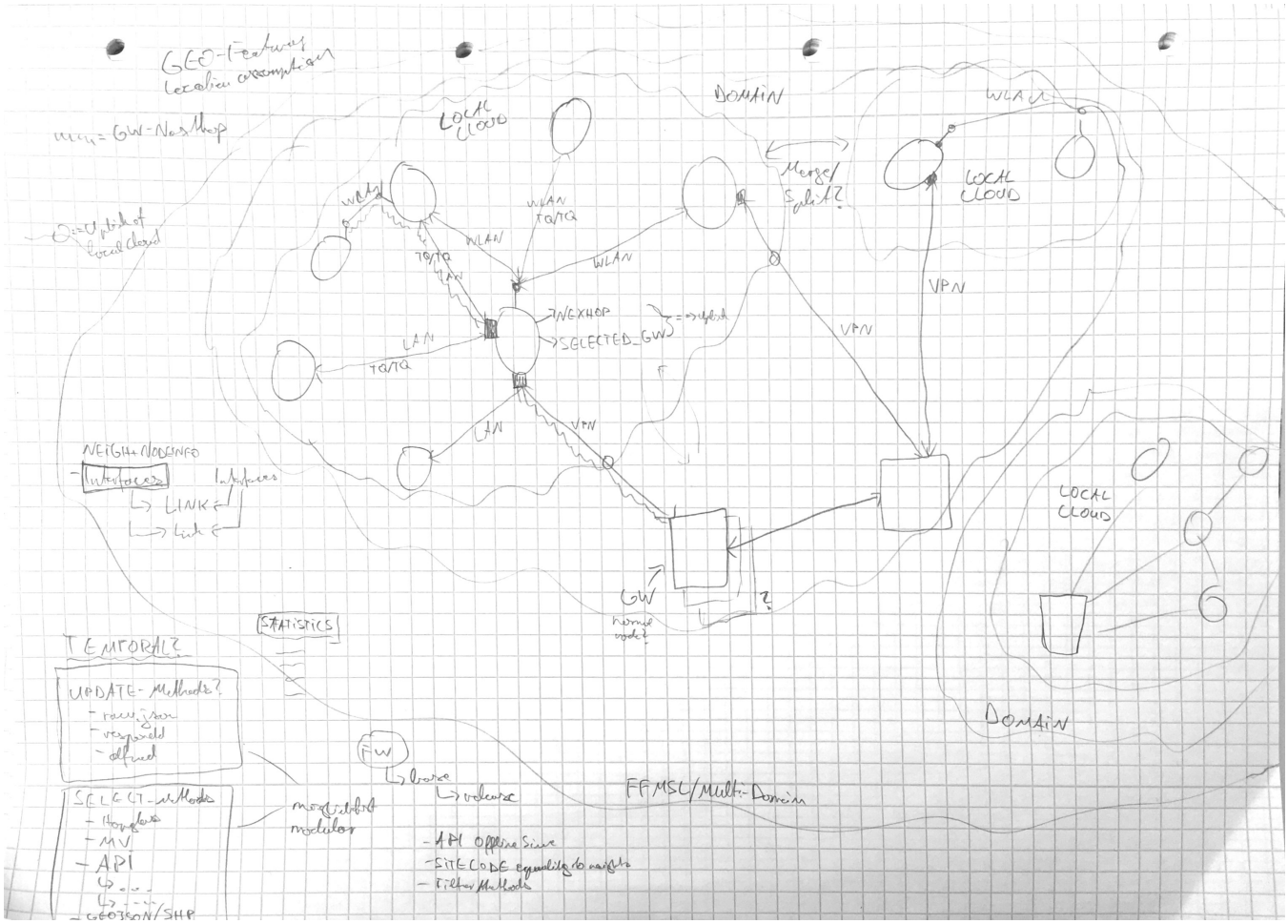
Optional wären folgende Eigenschaften wünschenswert:

- Temporale Datenerfassung (History/zeitliche Änderungen)
- Geospatiale Funktionen (z. B. alle Knoten innerhalb einer geometrischen Form abfragen)

Neben der eigentlichen Datenhaltung soll auch bereits eine Schnittstelle entwickelt werden. Diese soll einige, die für FFMS derzeit dringlichsten, Module enthalten. Die Entwicklung dieser Anwendung erfolgt aller Voraussicht nach in Python.

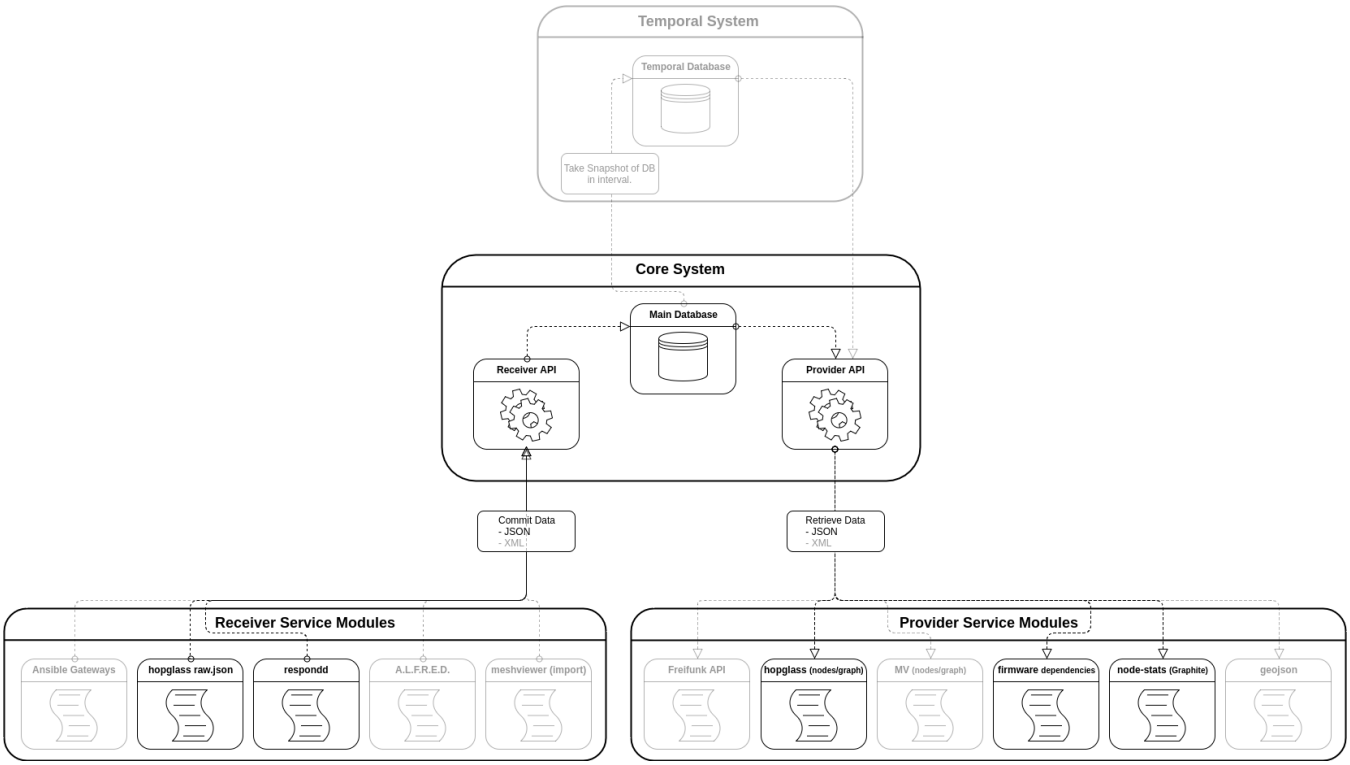
Entwurf

Konzeptionelle Ideen



Modularität

Die Entwicklung soll möglichst modular sein. Dazu wird an dieser Stelle eine mögliche Struktur der Module ausgearbeitet.

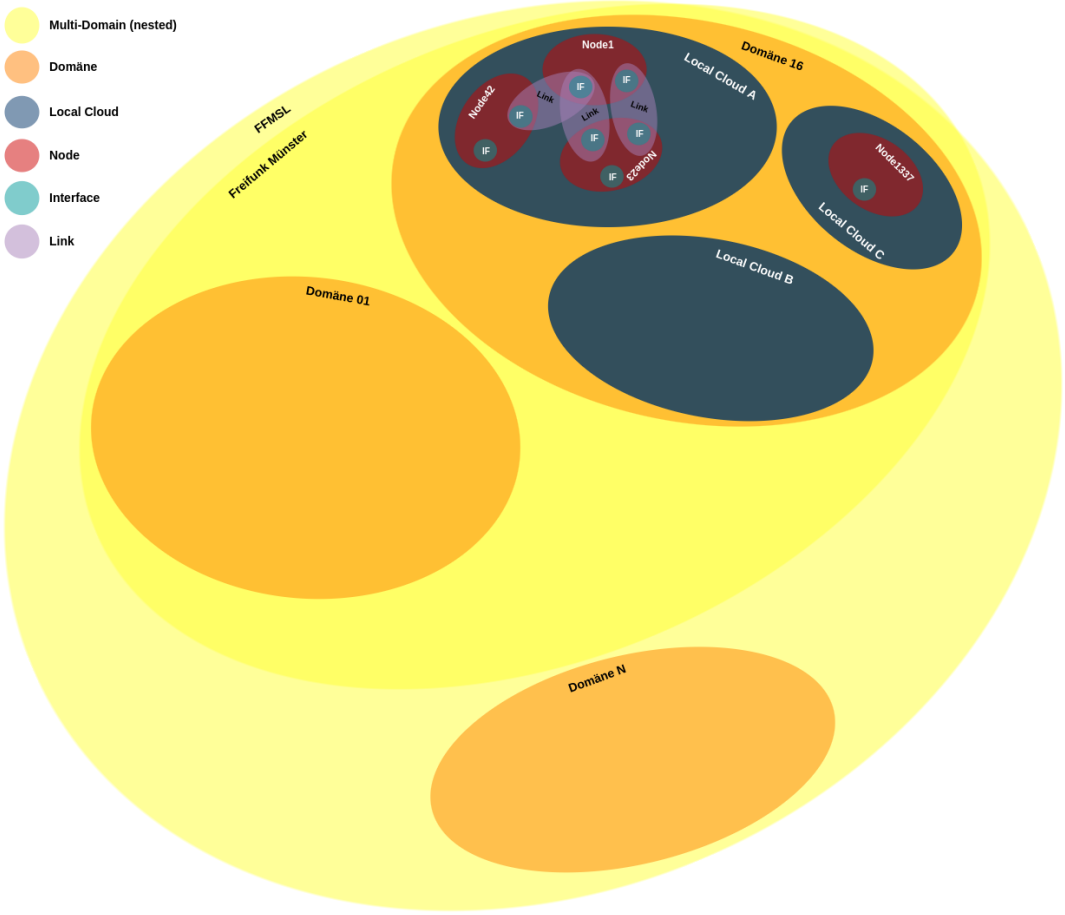


Anmerkung: Ausgegraute Punkte sind optional bzw. von niedriger Priorität.

Dieses Schaubild soll neben der Modularität ebenfalls verdeutlichen, dass Datenbank, APIs und die einzelnen Module jeweils unabhängig voneinander in unterschiedlichen Sprachen implementiert werden können. Dieses erhöht die möglichen Einsatzorte, sowie die Anzahl an potentiell Mitwirkenden bzw. Beitragenden zu diesem System.

Datenstrukturen

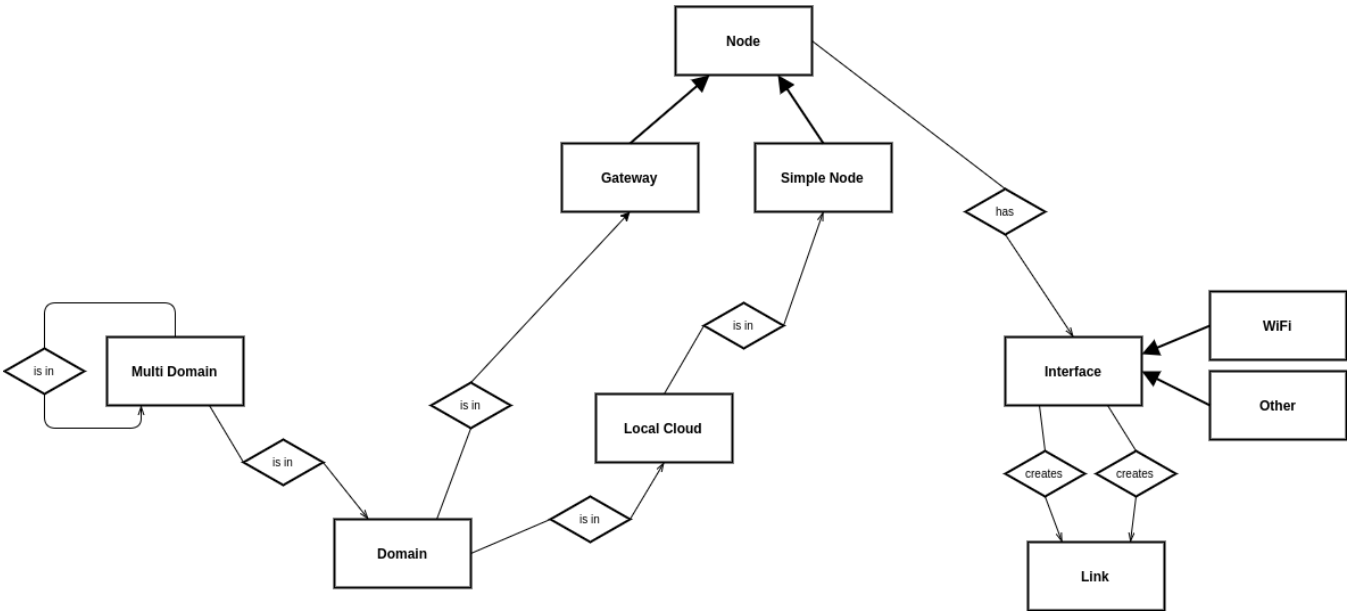
Venn Diagramm



Offene Fragen:

- Was ist mit den Gateways?
- Was ist mit den Links zu den Gateways?

Relationen



Datenbank

FIXME: Zunächst Dummy-Diagramme eingefügt. Werden noch ausgeführt.