09.12.2019 // Grundlegendes

Zuverlässigkeit ist Hauptziel; wenn BF_2H als Schließsystem für Türen o.ä eingesetzt werden soll kann ein Crash bedeuten das nur noch das Backup-Personal mit Schlüssel in das Labor kommen kann.

Der laufende Wartungsaufwand sollte null sein. Einmal aufgesetzt sollte das System möglichst ohne Unterbrechung laufen können. Datenquellen wie Datenbanken sollten darauf ausgelegt sein nicht korrumpierbar zu sein, Fehlerquellen wie nicht überprüfte dynamische Konfiguration die durch Schreibfehler o.ä. im späteren Betrieb Fehler aufwirft sollte so gut wie möglich im Vorhinein überprüft werden. Upgradepfade von Versionen zu neueren Versionen sollten klar sein. Ein Versionierungsschema wie Semver oder PVP sollte verwendet werden und Integrationstests sollten explizit darauf ausgelegt sein Änderungen in dokumentiertem Verhalten ohne Versionsänderung als Defekt zu deklarieren.

Das System muss erweiterbar sein; insbesondere Anbindung an bereits vorhandene Schließsysteme sollte möglich sein ohne das grosse Änderungen im Kern vorgenommen werden müssen. Hier bietet sich tendenziell ein System mit zur Laufzeit ladbaren Modulen an.

Anbindung an andere externe Systeme — z.b. Authentifizierung/Autorisierung über ActiveDirectory / LDAP / SAML oder Abrechnung über ERP wie Odoo — sollten auch möglich sein, aber die Anzahl der Schnittstellen die benötigt werden ist tendenziell wesentlich geringer also ist hier der Vorteil von anwendergeschriebenen Modulen weniger groß bzw. fehlende Zentralisierung kann die Qualität der Schnittstellen senken; i.e. eine einzelne generische Schnittstelle für LDAP und AD ist wesentlich sinnvoller als mehrere für jeweils die eine Instanz in den jeweiligen offenen Räumen.

Spezifische Technologien

Frontend nicht im Umfang dieses Dokuments.

Design Backend

Sprachwahl: Rust. Bestes Kosten/Nutzen-Verhältnis für stabile Software. Weniger populär als Java/C#/Python/PHP/C++/C, aber das ist nur dann relevant wenn die Software im Basar-Stil von sehr vielen Leuten entwickelt wird die alle nur sehr kleine Änderungen beitragen. Sehe ich hier nicht als gegeben, eher wahrscheinlich ist das 80-90% des Codes von einer Handvoll Entwicklern geschrieben wird und nur Fringe-Code von “Externen”

beigetragen wird.

Vorteile von Rust:

- Effiziente Programmiersprache
- Kompiliert zu einer dynamisch oder statisch verlinkten ELF
- Typsystem mit Typinferenz, statisch, stark & linear.
- Extrem gut dokumentiert, vor allem für Anfänger viele Möglichkeiten die Sprache umfassend zu lernen.
- Fehlerbehandlung über Summentypen (Some/None, Ok/Err) anstatt über Exceptions*
  - Macht Fehlerbehandlung extrem resilient da alle Behandlung explizit passieren muss.
  - Alle Fehler die eine Funktion generieren kann sind im Funktionsprototypen festgelegt
    - Probleme der Art eine neu hinzugekommen Exception wird (noch) nicht behandelt *können* nicht passieren.
- Flexibles Futures-System und von Anfang auf Multithreading ausgelegtes Design machen es sehr einfach hoch-parallelisierte Anwendungen zu schreiben.
  - Für uns sehr relevant da wir eine solche schreiben wollen.
- Guter Library-Support für unsere Anwendungsfälle.
- Sollte Bedarf bestehen ist es sehr einfach Support für Skriptsprachen wie Python oder Lua einzubauen

(* Rust hat “Exceptions” in der Form von `panics`. Diese sind aber anders zu sehen als Exceptions in z.B. C++ oder Python weil sie nur für schwere und nicht behebbare Fehler gedacht sind und auch nicht innerhalb einen Threads abgefangen werden können)

Kommunikation

Kommunikation zwischen Komponenten:

(P2P steht in diesem Kontext fuer Point-to-Point also Unicast, P2MP steht fuer Point-to-MultiPoint, also Multicast, nicht fuer Peer-to-Peer im Stil von BitTorrent o.ae.)

- Schliessystem(e) mit Backend
  - Stil: P2P & P2MP (Backend->Clients)
  - Bidirektionales message passing
    - RPC von einzelnen Clients zum zentralen Server
    - Statusabfragen vom zentralen Server an Clients

    - P2MP für Broadcasts oder Multicast vom Hauptprogramm zu Schliesssystemen
        - Notifications vom zentralen Server an mehrere Clients
            - Beispiel: Feierabend, bestimme Maschinen in 30min herunterfahren
        - Kann als PubSub gebaut werden, evtl allerdings vom Server diktiert sinnvoller.
            - Bei PubSub müssen die Clients selber wissen ob diese Art Nachricht
              für sie relevant ist. In unserem Fall ist es aber evtl. sinnvoller wenn der Server das eigenmächtig entscheiden kann.
        - Auch für Broadcasts bei z.B. Neustart des zentralen Servers
    - Rückwärtskompatibilität im Protokoll sehr wichtig. Hinzufügen eines Feldes für einen neuen Typ Client sollte es nicht erzwingen das auf allen anderen Clients Code-Updates erforderlich werden.
        - Schnittstellenbeschreibungssysteme (IDS) wie Google Protocol Buffers, Apache
          Thrift, Cap’n’Proto, … legen darauf besonders Augenmerk, ist also in diesen einfach.
        - ASN.1 mit PER und BER erlaubt das gleiche, kann aber einen speziell dafür
          angepassten Parser benötigen.
- Frontend mit Backend
    - Stil: p2p
    - Bidirektionales Message passing
        - Notifications vom Backend zum Frontend
        - RPC vom Frontend zum Backend
    - Netzwerk-basierte IPC damit auch lokale Frontends (GUI/TUI) verwendet werden
      können
        - Besonders ein CLI ist für Automatisierung sehr wichtig
    - Da es sehr wahrscheinlich mehr als ein Frontend geben wird sollte auch hier ein IDS verwendet werden.
- Backend mit externen Diensten
    - Dienst-spezifische Protokolle & Aufbau (z.B. LDAP, HTTP(S), …)

Sinnvolle Protokolle:

- reines TCP
- ØMQ
- MQTT (v5)
- ?

Netzwerkabstraktion wäre schön, aber HTTP löst die gegebenen Probleme schlechter als reines TCP, sollte also nicht verwendet werden.

Websockify für Web-basierte Frontends?

alt.: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events

Verschlüsselung / Sicherheit

Backend mit Schliesssystemen

TLS ist ein extrem komplexes Protokoll, kann also vor allem für embedded Plattformen zu schwer werden. Zusätzlich bietet TLS uns keine Vorteile; die wichtigsten Eigenschaften des TLS-Stacks sind Aushandlung von Algorithmen und Zertifikatsvalidierung via PKI. Alternative: Nachrichten werden mit festgelegtem Algorithmus (z.B. Salsa20/Poly1305 oder AES256-GCM) verschluesselt. Beim erstmaligen Einrichten einen embedded Clients wird ein System-Key generiert, dessen public key auch dem Backend-Server bekannt ist und der für Authentifizierung des Clients gegenueber des Backend-Server genutzt wird.

Backend mit Frontends

Frontends werden auf wesentlich leistungsstärkeren Plattformen aufgesetzt sein. Vor allem bei lokalen Frontends ist TLS sehr sinnvoll.

Design Schliesssysteme

Primär Leistungsschwache, auf Kosten optimierte Hardware.
Oft gesehen: ESP8266, Arduino

Vor allem bei ESP8266 ist ein System wie NodeMCU sinnvoll. Das Protokoll zum Server / Crypto kann in C/C++/… implementiert werden, die Maschinen-spezifische Handhabung dann in Lua.

Misc.

- Module wieder upstreamen
- Konfigurationswebinterface

Moduldesign

Spezifisch Kommunikationsmodule mit Schliesssytemen:

- Aus Stabilitätsgründen nicht im selben Prozess wie Server
  - Kommunikation über UNIX socket oder named pipe
  - Protokoll via IDS
  - Support für Python & Rust am sinnvollsten, Lua evtl. auch hilfreich
    - Grundsätzlich kann aber jede Sprache verwendet werden die auf UNIX sockets oder named pipes zugreifen kann
- Falls später komplexere Module benötigt werden neuer Modultyp die als .so zur Laufzeit geladen werden
  - Haben direkten Zugriff auf alle interen APIs, können aber den Server crashen
- Von anderen geschriebene Module sollten von Anfang an zentral gesammelt werden damit alle davon profitieren können.
- Was spricht dagegen das Protokoll fuer Referenzclients fuer Module zu verwenden?
  - Dann zwingend notwendig: Mehrere Clients ueber eine Verbindung. Tendenziell aber eh sinnvoll.

<!- vim: set spelllang=de: ->

Version #2
Erstellt: 13 Oktober 2024 01:07:01 von Mario Voigt (Stadtfabrikanten e.V.)
Zuletzt aktualisiert: 14 Dezember 2024 18:23:07 von Mario Voigt (Stadtfabrikanten e.V.)