FabAccess Server-Client Konzept

Stream Initiierung

Bei einer Sitzung gibt es zwei Parteien: Die einleitende Stelle und die empfangende Stelle. Diese Terminologie bezieht sich nicht auf den Informationsfluss, sondern vielmehr auf die Seite, die eine Verbindung eröffnet bzw. auf die Seite, die auf Verbindungsversuche wartet. In dem derzeit angedachten Anwendungsfall ist die initiierende Instanz ...

- a) ein Client (d.h. ein interaktives oder Batch/automatisiertes Programm), der versucht, auf die eine oder andere Weise mit einem Server zu interagieren,
- b) ein Server, der versucht, Informationen mit/von einem anderen Server auszutauschen/anzufordern (d.h. Föderation). Die empfangende Einheit ist jedoch bereits ein Server.

Außerdem ist die Anzahl und Art der Clients vielfältiger und weniger aktuell als die der Server. Daraus ziehen wir folgende Schlüsse:

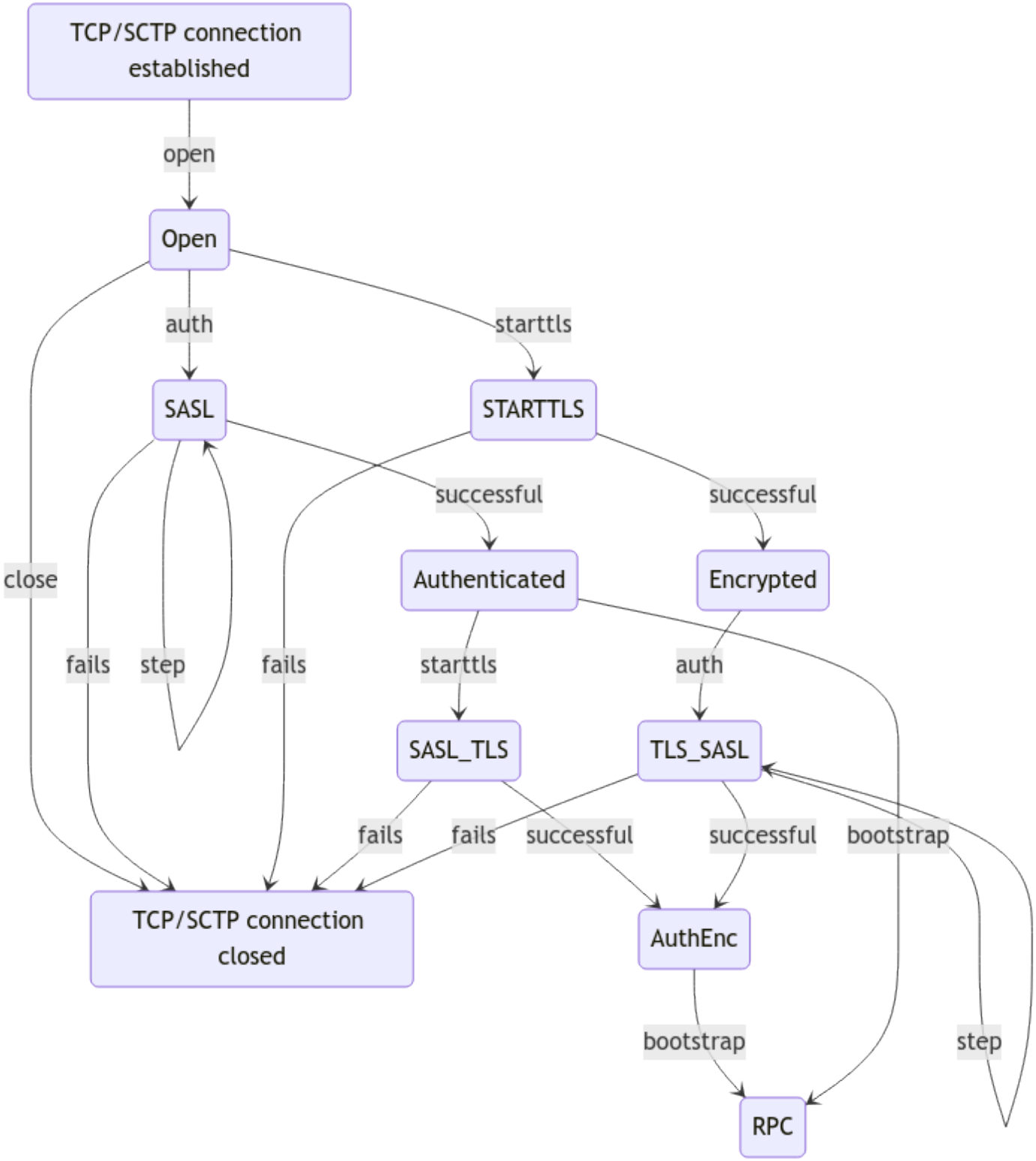
- Es ist wahrscheinlicher, dass Clients eine veraltete Version des Kommunikationsprotokolls implementieren.
- Der Ort für Rückwärtskompatibilität sollten die Server sein.
- Daher sollte der Client (auslösende Einheit) zuerst die erwartete API-Version senden, auf deren Grundlage der Server dann entscheidet, mit welcher API-Version er antworten wird.

Stream Negotiation

Da die empfangende Stelle für die von ihr kontrollierten Ressourcen verantwortlich ist, stellt sie Bedingungen für die Verbindung entweder als Client oder als föderierender Server. Zumindest muss sich jede einleitende Stelle gegenüber der empfangenden Stelle authentifizieren, bevor sie weitere Aktionen durchführt oder Informationen anfordert. Eine empfangende Stelle kann aber auch andere Merkmale verlangen, z. B. eine Verschlüsselung auf der Transportschicht. Zu diesem Zweck informiert eine empfangende Stelle die einleitende Stelle über Merkmale, die sie von der einleitenden Stelle benötigt, bevor sie weitere Aktionen durchführt, sowie über Merkmale, die freiwillig ausgehandelt werden, aber die Qualität des Datenstroms verbessern können (z. B. Nachrichtenkompression).

Unterschiedliche Bedingungen machen eine Verhandlung erforderlich. Da die Funktionen möglicherweise eine strenge Reihenfolge erfordern (z. B. Verschlüsselung vor Authentifizierung), muss die Aushandlung in mehreren Schritten erfolgen. Weitere Einschränkungen ergeben sich daraus, dass einige Funktionen erst angeboten werden können, nachdem andere eingerichtet wurden (z. B. SASL-Authentifizierung wird erst nach der Verschlüsselung verfügbar, der EXTERNAL-Mechanismus ist nur für lokale Sockets oder Verbindungen verfügbar, die ein Zertifikat bereitstellen).

Verbindungsstatus

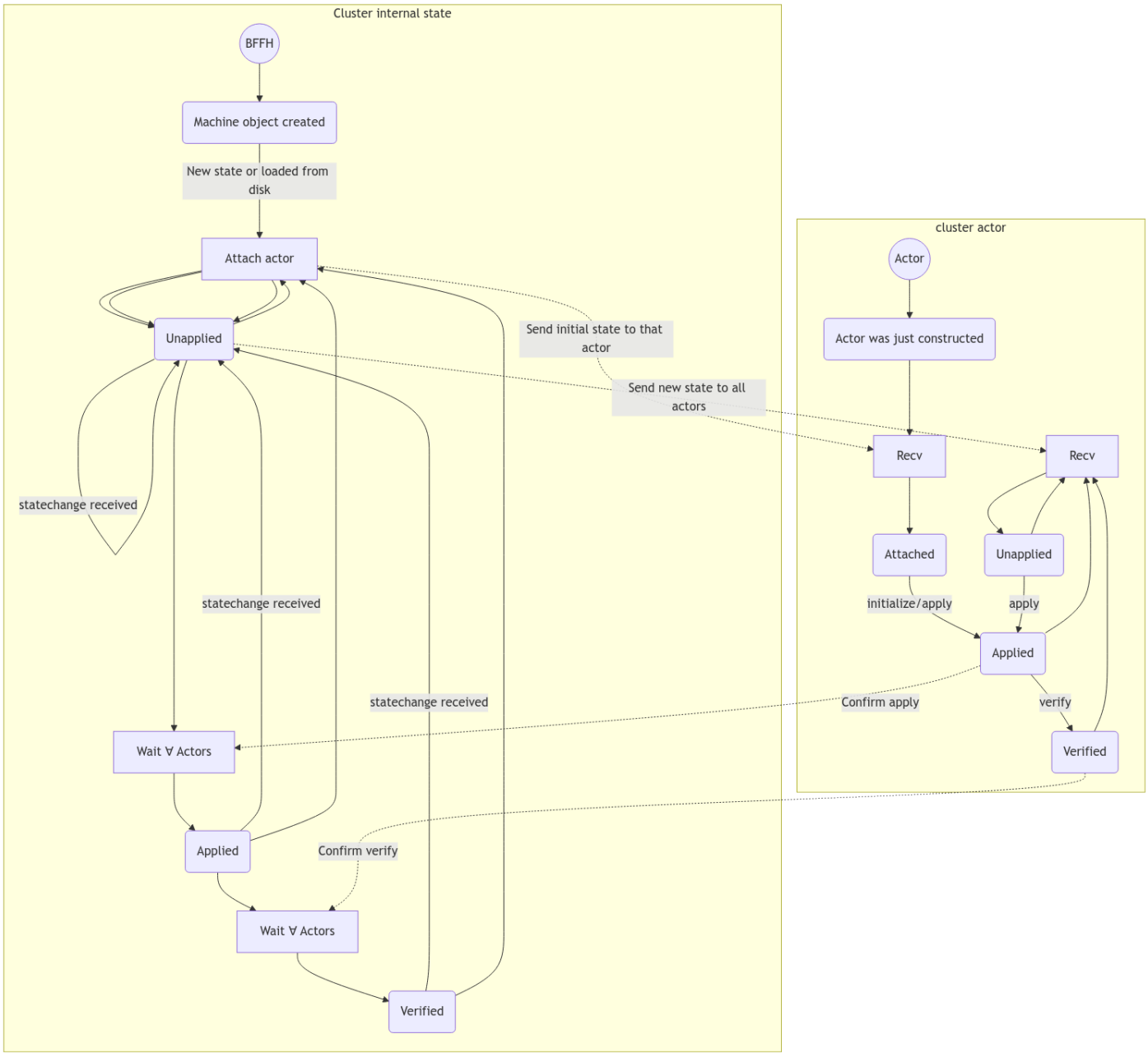


als Mermaid

```
stateDiagram
    state "TCP/SCTP connection established" as Establish
    state "TCP/SCTP connection closed" as Closed
```

```
    Open
    SASL
    Authenticated
    STARTTLS
    Encrypted

    Establish --> Open:open

    Open --> Closed:close

    Open --> SASL:auth
    SASL --> SASL:step
    %% Authentication fails
    SASL --> Closed:fails
    %% Authentication succeeds
    SASL --> Authenticated:successful

    Open --> STARTTLS:starttls
    %% TLS wrapping succeeds
    STARTTLS --> Encrypted:successful
    %% TLS wrapping fails
    STARTTLS --> Closed:fails

    Authenticated --> SASL_TLS:starttls
    SASL_TLS --> Closed:fails
    SASL_TLS --> AuthEnc:successful

    Encrypted --> TLS_SASL:auth
    TLS_SASL --> TLS_SASL:step
    TLS_SASL --> Closed:fails
    TLS_SASL --> AuthEnc:successful

    %% Only authenticated connections may open RPC. For "unauth", use the `Anonymous` SASL
method.
    AuthEnc --> RPC:bootstrap
    Authenticated --> RPC:bootstrap
```

Andere Stati (Cluster Kommunikation BFFH und Aktoren)



als Mermaid

```
graph

    subgraph First[Cluster internal state]
        start((BFFH))
        created(Machine object created)
```

```
    start --> created

    created -- New state or loaded from disk --> attach
    attach[Attach actor]
    unapplied(Unapplied)
    applied(Applied)
    verified(Verified)

    wait_apply[Wait ∀ Actors]
    wait_verify[Wait ∀ Actors]

    unapplied --> wait_apply
    wait_apply --> applied
    applied --> wait_verify
    wait_verify --> verified

    applied -- statechange received --> unapplied
    verified -- statechange received --> unapplied
    unapplied -- statechange received --> unapplied

    unapplied --> attach
    attach --> unapplied
    applied --> attach
    attach --> unapplied
    verified --> attach
    attach --> unapplied
end


subgraph Second[cluster actor]
    actor_start((Actor))
    actor_fresh(Actor was just constructed)
    actor_start --> actor_fresh

    actor_attached(Attached)
    actor_unapplied(Unapplied)
    actor_applied(Applied)
    actor_verified(Verified)
```

```
        wait_initial[Recv]
        wait_state[Recv]

        actor_fresh --> wait_initial
        wait_initial --> actor_attached

        actor_attached -- initialize/apply --> actor_applied
        actor_unapplied -- apply --> actor_applied
        actor_applied -- verify --> actor_verified

        actor_unapplied --> wait_state
        actor_applied --> wait_state
        actor_verified --> wait_state

        wait_state --> actor_unapplied
    end

    attach -. Send initial state to that actor .-> wait_initial
    unapplied -. Send new state to all actors .-> wait_state
    actor_applied -. Confirm apply .-> wait_apply
    actor_verified -. Confirm verify .-> wait_verify
```

> **Hinweis:** Die Mermaid-Diagramme sind mit https://mermaid.live gerendert und als PNG exportiert und hier importiert worden, da BookStack keinen integrierten Mermaid Renderer besitzt.

Version #10
Erstellt: 2025-02-20 13:57:06 CET von Mario Voigt (Stadtfabrikanten e.V.)
Zuletzt aktualisiert: 2025-02-20 15:24:03 CET von Mario Voigt (Stadtfabrikanten e.V.)