AGENTSPROTOCOL

Technische Spezifikation & Entwicklerdokumentation

Version 1.0 — April 2026

Fatih Dinc | fatdinhero@gmail.com | agentsprotocol.org

Contents

1 Architekturuebersicht

Das AgentsProtocol-Netzwerk besteht aus drei logischen Schichten, die unabhaengig voneinander skaliert und ausgetauscht werden koennen. Die **Validierungsschicht** empfaengt Claims, berechnet Scores und erzeugt Zero-Knowledge-Beweise. Die **Konsensschicht** ordnet akzeptierte Bloecke via GHOSTDAG in einen gerichteten azyklischen Graphen. Die **Abfrageschicht** ermoeglicht leichten Clients die Verifikation ohne vollstaendige Blockchain-Replikation.

Empfohlener Tech-Stack: Rust (Validator-Client), RocksDB (lokaler Speicher), rust-libp2p (P2P), risc0-zkvm oder Nexus SDK (Zero-Knowledge), ONNX Runtime + Sentence Transformers (Embeddings), ed25519-dalek + sha2 (Kryptographie).

1.1 Modul-Struktur (Referenz-Client)

```
validator/
|-- main.rs
|-- network/          # P2P (libp2p), RPC-Endpunkte
|-- consensus/        # GHOSTDAG, Blockverarbeitung
|-- validation/       # S_con, WiseScore, Psi-Test
|-- zk/               # zkVM-Interface (Nexus / RISC Zero / SP1)
|-- storage/          # DAG, Claims, Bloecke (RocksDB)
|-- stake/            # Token-Staking-Logik
`-- config/           # Protokollparameter, Versionierung
```

Listing 1: Empfohlene Verzeichnisstruktur des Validator-Clients

2 Claim-Datenformat (JSON Schema)

Jeder Claim wird als JSON-Objekt mit der folgenden Struktur eingereicht. Das Schema ist Teil der Protokollversion und wird als Hash im Genesisblock verankert.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "required": [
    "protocol","version","type","id",
    "timestamp","submitter","signature","payload"
  ],
  "properties": {
    "protocol":  { "const": "agentsprotocol" },
    "version":   { "type": "string", "pattern": "^[0-9]+\\.[0-9]+$" },
    "type":      { "const": "claim" },
    "id":        { "type": "string",
                   "description": "SHA-256 hash des serialisierten
                      payload" },
    "timestamp": { "type": "string", "format": "date-time" },
    "submitter": { "type": "string",
                   "description": "Ed25519-Publick-Key (hex)" },
    "signature": { "type": "string",
                   "description": "Ed25519-Signatur ueber id" },
    "payload": {
      "type": "object",
```

```
      "required": ["statement","entities","predicate","value"],
      "properties": {
        "statement": { "type": "string" },
        "entities": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["name","type"],
            "properties": {
              "name": { "type": "string" },
              "type": { "type": "string" },
              "uri":  { "type": "string", "format": "uri" }
            }
          }
        },
        "predicate": { "type": "string" },
        "value":     { "type": "object" },
        "evidence": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["type","uri"],
            "properties": {
              "type":      { "type": "string" },
              "uri":       { "type": "string", "format": "uri" },
              "timestamp": { "type": "string", "format": "date-time" }
            }
          }
        },
        "context": {
          "type": "object",
          "properties": {
            "domain":          { "type": "string" },
            "knowledgeCorpus": { "type": "string", "format": "uri",
                                 "description": "IPFS CID des
                                    Wissenskorpus k" }
          }
        }
      }
    }
  }
}
```

Listing 2: Claim JSON Schema (agentsprotocol-claim-v1.0.json)

2.1 Beispiel-Claim

```
{
  "protocol": "agentsprotocol",
  "version":  "1.0",
  "type":     "claim",
  "id":       "0x7a9f3c...",
  "timestamp":"2026-04-18T14:30:00Z",
  "submitter":"0x1234abcd...",
```

```
  "signature":"0xef56...",
  "payload": {
    "statement": "Die Temperatur in Berlin betraegt 20 Grad Celsius.",
    "entities": [
      { "name": "Berlin", "type": "location", "uri": "wikidata:Q64" }
    ],
    "predicate": "hatTemperatur",
    "value": { "amount": 20, "unit": "Celsius" },
    "evidence": [
      {
        "type": "sensor",
        "uri":  "ipfs://QmT...",
        "timestamp": "2026-04-18T14:25:00Z"
      }
    ],
    "context": {
      "domain": "Wetter",
      "knowledgeCorpus": "ipfs://QmK..."
    }
  }
}
```

Listing 3: Beispiel: Temperatur-Claim

3 Berechnung des semantischen Konsistenz-Scores S_{con}

3.1 Algorithmus

Der S_{con}-Score wird deterministisch in drei Schritten berechnet. Die verwendeten Modelle und der Schwellwert τ sind Teil der Protokollversion und als Hash im Blockheader verankert.

Schritt 1 – Extraktion: Aus dem Claim-Text wird mittels eines vortrainierten Satz-Transformers (z. B. `sentence-transformers/all-MiniLM-L6-v2`) ein Embedding-Vektor $\mathbf{v}_A \in \mathbb{R}^d$ erzeugt.

Schritt 2 – Abruf: Aus dem Wissenskorpus κ (adressiert ueber IPFS CID) werden alle Fakten abgerufen, die dieselben Entitaeten betreffen. Deren Embeddings $\{\mathbf{v}_\kappa^{(1)}, \ldots, \mathbf{v}_\kappa^{(m)}\}$ werden gemittelt zu $\bar{\mathbf{v}}_\kappa$.

Schritt 3 – Aehnlichkeitsmass:

$$S_{\text{con}}(A) = \max\left(0,\ \frac{\cos\!\left(\mathbf{v}_A, \bar{\mathbf{v}}_\kappa\right) - \tau}{1 - \tau}\right), \quad \tau \in [0, 1)$$

3.2 Pseudocode (Python)

```python
def compute_s_con(
    claim_text: str,
    knowledge_corpus_cid: str,
    embed: callable,
    tau: float = 0.7
) -> float:
    v_claim = embed(claim_text)
    corpus  = ipfs_cat(knowledge_corpus_cid)
    facts   = retrieve_facts(corpus, claim_text)  # Entitaeten-Lookup
```

```
    if not facts:
        return 0.0
    v_mean  = np.mean([embed(f) for f in facts], axis=0)
    cos_sim = np.dot(v_claim, v_mean) / (
              np.linalg.norm(v_claim) * np.linalg.norm(v_mean))
    return max(0.0, (cos_sim - tau) / (1.0 - tau))
```

Listing 4: Referenzimplementierung S_con

4 Kontrollaufgaben & Ψ-Test

4.1 Kontrollaufgaben-Set

Das Kontrollaufgaben-Set $\{D_1, \ldots, D_k\}$ mit festgelegten Referenz-Scores $S^*(D_j)$ wird per Governance-Abstimmung beschlossen und als Hash im Genesisblock verankert. Neue Aufgaben erfordern eine Karenzzeit von mindestens vier Wochen.

```
{
  "controlSetId": "v1",
  "genesisHash":  "0xabc123...",
  "claims": [
    {
      "id": "ctrl-001",
      "statement": "Die Hauptstadt Frankreichs ist Paris.",
      "expectedScore": 1.0
    },
    {
      "id": "ctrl-002",
      "statement": "Wasser kocht bei 100 Grad Celsius auf Meereshoehe.",
      "expectedScore": 0.98
    }
  ]
}
```

Listing 5: Kontrollaufgaben-Set v1 (Auszug)

4.2 Gewichtete Ψ-Statistik

Jeder Validator i erzeugt den Fehlervektor

$$\mathbf{e}_i = \big(|S_i(D_j) - S^*(D_j)|\big)_{j=1}^{k}.$$

Die gewichtete Ψ-Statistik mit $w_i = \sqrt{s_i}$ (Stake des Validators i) ist:

$$\Psi = 1 - \frac{\sum_{i<j} w_i\, w_j\, |\rho(\mathbf{e}_i, \mathbf{e}_j)|}{\sum_{i<j} w_i\, w_j}$$

```
from math import sqrt
from scipy.stats import pearsonr

def compute_psi(
```

```
    error_vectors: list,
    stakes: list
) -> float:
    n = len(error_vectors)
    if n < 2:
        return 1.0
    weights = [sqrt(s) for s in stakes]
    w_sum, corr_sum = 0.0, 0.0
    for i in range(n):
        for j in range(i + 1, n):
            w = weights[i] * weights[j]
            r = abs(pearsonr(error_vectors[i], error_vectors[j])[0])
            corr_sum += w * r
            w_sum    += w
    if w_sum == 0:
        return 0.0
    return max(0.0, min(1.0, 1.0 - corr_sum / w_sum))
```

Listing 6: Referenzimplementierung Psi-Test

5 Blockstruktur & DAG-Konsens

5.1 Blockaufbau

Jeder Block besteht aus drei Teilen. Der **Header** enthaelt: Protokollversion, Hashes aller Elternbloecke (GHOSTDAG), Merkle-Root der enthaltenen Claims, Hash des zkVM-Beweises, Zeitstempel, Ψ-Wert und kumuliertes Gewicht. Der **Body** enthaelt: die Liste der validierten Claims, eine Coinbase-Transaktion (Validator-Belohnung) sowie Gebuehren-Transaktionen. Der **zk-Proof** ist ein succinct Beweis der korrekten S_{con}-Berechnung.

5.2 Blockgewicht und kanonischer Pfad

$$\text{Gewicht}(B) = \Psi_B \cdot \sum_{A \in B} S_{\text{con}}(A)$$

Die kanonische Kette ist der Pfad durch den DAG mit dem hoechsten kumulierten Gewicht. Bei konkurrierenden Pfaden setzt sich derjenige mit hoherem Gewicht durch.

```
def block_weight(block) -> float:
    return block.psi * sum(c.s_con for c in block.claims)

def canonical_path(dag) -> list:
    # Waehlt den Pfad mit hoechstem kumuliertem Gewicht
    return dag.heaviest_path()
```

Listing 7: Blockgewicht-Berechnung

6 Zero-Knowledge-Integration (zkVM)

Das Protokoll ist zkVM-agnostisch. Als Referenzimplementierung dient die Nexus zkVM; jede RISC-V-basierte zkVM mit succinct proofs ist kompatibel (z. B. RISC Zero, SP1).

```
// Gaestprogramm -- laeuft in der zkVM
fn main() {
    let claim:  Claim           = read_input();
    let corpus: KnowledgeCorpus = read_input();
    let s_con = compute_s_con(&claim, &corpus);
    commit(&s_con);  // oeffentlicher Output
}
```

Listing 8: Gastprogramm (Validator-Seite, Rust)

```
// Host -- laeuft ausserhalb der zkVM
let receipt = prover
    .prove(GUEST_ELF, (claim, corpus))
    .expect("Proof-Erzeugung fehlgeschlagen");
receipt.verify(GUEST_ID).expect("Beweis ungueltig");
let s_con: f32 = receipt.journal.decode().unwrap();
// s_con und receipt.hash gehen in den Blockheader
```

Listing 9: Host-Seite (Validator erzeugt und veroeffentlicht Beweis)

Leichte Clients verifizieren ausschliesslich den kompakten Beweis (ca. 200 KB) und muessen die Berechnung selbst nicht wiederholen.

7 Netzwerkprotokoll (P2P)

Transport: libp2p (TCP/QUIC) mit noise-Protokoll fuer Verschluesselung und yamux fuer Multiplexing.

PubSub-Themen (gossipsub):

darkblue	
/agentsprotocol/claims/1.0.0	Neue Claims (JSON, signiert)
lightgray	Neue Bloecke inkl. zk-Beweis
/agentsprotocol/blocks/1.0.0	
/agentsprotocol/control/1.0.0	Kontrollaufgaben-Updates
lightgray	Peer-Discovery
/agentsprotocol/peers/1.0.0	

RPC-Endpunkte (gRPC / JSON-RPC):

```
SubmitClaim(Claim)           -> ClaimId
GetClaim(ClaimId)            -> ClaimWithProof
GetBlockHeader(height)       -> BlockHeader
GetBlock(height)             -> Block
GetValidatorStatus(pubkey)   -> ValidatorInfo
GetNetworkParams()           -> ProtocolParams
```

Listing 10: API-Endpunkte

8 Akzeptanzregel

Ein Block B wird genau dann in den DAG aufgenommen, wenn:

$$\frac{1}{|B|}\sum_{A\in B} S_{\text{con}}(A) \;\geq\; \theta_{\min} \quad \text{und} \quad \Psi_B \;\geq\; \Psi_{\min} \quad \text{und} \quad \pi_B \text{ verifiziert.}$$

Die Parameter $\theta_{\min}$ (Standardwert: 0.6) und $\Psi_{\min}$ (Standardwert: 0.7) sind Governance-Parameter und koennen durch Token-Abstimmung angepasst werden.

9 Beitragen & Mitwirken

Das Projekt befindet sich in der Spezifikationsphase. Beitraege sind in folgenden Bereichen besonders willkommen.

Phase 1 (sofort): Implementierung des Claim-Parsers und der S_{con}-Bibliothek in Rust oder Python. Erstellung von Unit-Tests fuer das Claim-Schema.

Phase 2: Simulation des Ψ-Tests mit synthetischen Validator-Fehlervektoren. Implementierung des GHOSTDAG-Konsensalgorithmus.

Phase 3: Aufbau eines lokalen Testnets (ohne zkVM). Integration einer RISC-V-basierten zkVM.

Phase 4: Vollstaendige Konsensimplementierung, Sicherheitsaudit, Testnet-Launch.

Einstiegspunkte: GitHub-Issues mit dem Label `good first issue` im Repository `agentsprotocol/sp`

Ressourcen

Whitepaper: agentsprotocol.org/whitepaper
Repository: github.com/agentsprotocol/specification
Kontakt: fatdinhero@gmail.com