```
// ═══════════════════════════════════════════════════════════════════════════
// MODULE 1 : temporal_code_generator.ts
// Génération et vérification du code temporel
// Tourne côté serveur de nœud Aequitaverse
// ═══════════════════════════════════════════════════════════════════════════

import crypto from 'crypto';

// ── Types ──────────────────────────────────────────────────────────────────

interface CodeTemporel {
  codeAffichable:   string;     // AEQ-2026-K7M4-XP9R-2B5N-W3QF
  codeRaw:          Uint8Array;  // 36 octets bruts
  timestampMs:      number;     // Milliseconde de génération
  expirationMs:     number;     // timestampMs + 15 minutes
  hashBloc:         string;     // Hash du dernier bloc blockchain
  hashLSHSession:   string;     // Lié au profil LSH soumis
  defiComportemental: DefiComportemental;
  signatureServeur: string;     // Signature HMAC du serveur
}

interface DefiComportemental {
  id:          string;
  instruction: string;   // Affiché à l'utilisateur
  type:        TypeDefi;
  dureeSecondes: number;
  hashAttendu: string;   // Hash du geste attendu (pour vérification)
}
```

```
type TypeDefi =
  | 'GESTE_MAIN'
  | 'ROTATION_TETE'
  | 'LECTURE_CHIFFRE'
  | 'CLIGNEMENT'
  | 'SOURIRE';

interface SessionCeremonie {
  sessionId:       string;
  codeTemporel:    CodeTemporel;
  lshHashSession:  string;
  statut:          StatutSession;
  tentatives:      number;
  ipAdresse:       string;  // Anti-spam, pas stocké après vérification
  timestamps: {
    creation:      number;
    debutFilmage?: number;
    finFilmage?:   number;
    attestation1?: number;
    attestation2?: number;
  };
}

type StatutSession =
  | 'CODE_GENERE'
  | 'FILMAGE_EN_COURS'
  | 'HASH_VIDEO_SOUMIS'
  | 'TEMOIN_1_ATTESTE'
  | 'TEMOINS_COMPLETES'
```

```
  | 'EXPIREE'
  | 'FRAUDULEUSE';

// ── Défis comportementaux disponibles ──────────────────────

const DEFIS: DefiComportemental[] = [
  {
    id: 'D01',
    instruction: 'Levez la main droite puis touchez votre nez',
    type: 'GESTE_MAIN',
    dureeSecondes: 5,
    hashAttendu: 'hash:geste:main_droite_puis_nez'
  },
  {
    id: 'D02',
    instruction: 'Faites "non" de la tête trois fois lentement',
    type: 'ROTATION_TETE',
    dureeSecondes: 5,
    hashAttendu: 'hash:geste:rotation_tete_non_x3'
  },
  {
    id: 'D03',
    instruction: 'Clignez des yeux deux fois délibérément',
    type: 'CLIGNEMENT',
    dureeSecondes: 4,
    hashAttendu: 'hash:geste:clignement_x2'
  },
  {
    id: 'D04',
```

```
    instruction: 'Souriez, puis redevenez neutre',
    type: 'SOURIRE',
    dureeSecondes: 4,
    hashAttendu: 'hash:geste:sourire_puis_neutre'
  },
  {
    id: 'D05',
    instruction: 'Levez les deux sourcils simultanément',
    type: 'GESTE_MAIN',
    dureeSecondes: 3,
    hashAttendu: 'hash:geste:sourcils_hausses'
  },
];

// ── Générateur de code temporel ──────────────────────────────

export class CodeTemporelGenerator {

  private readonly cleHMAC: Buffer;
  private readonly DUREE_VALIDITE_MS = 15 * 60 * 1000; // 15 minutes
  private readonly MAX_TENTATIVES    = 3;

  // Sessions actives en mémoire (pas en base — anti-traçage)
  private sessionsActives = new Map<string, SessionCeremonie>();

  constructor(cleHMAC: string) {
    this.cleHMAC = Buffer.from(cleHMAC, 'hex');
  }
```

```
// ── Générer un nouveau code temporel ──────────────────────

async genererCode(
  lshHashSession: string,   // Hash LSH du profil soumis
  hashDernierBloc: string,  // Hash du dernier bloc blockchain
  ipAdresse: string
): Promise<SessionCeremonie> {

  const now       = Date.now();
  const sessionId = crypto.randomBytes(16).toString('hex');
  const nonce     = crypto.randomBytes(8);

  // ── Construire le code brut (36 octets) ─────────────────
  const codeRaw = Buffer.alloc(36);

  // [0-7]   timestamp en millisecondes (BigInt 64 bits)
  const tsBuf = Buffer.alloc(8);
  tsBuf.writeBigInt64BE(BigInt(now));
  tsBuf.copy(codeRaw, 0);

  // [8-15]  premiers 8 octets du hash du dernier bloc
  Buffer.from(hashDernierBloc.replace('sha3:', ''), 'hex')
    .slice(0, 8).copy(codeRaw, 8);

  // [16-23] premiers 8 octets du hash LSH de session
  Buffer.from(lshHashSession, 'hex')
    .slice(0, 8).copy(codeRaw, 16);

  // [24-31] nonce aléatoire
```

```
nonce.copy(codeRaw, 24);

// [32-35] checksum CRC32 des 32 premiers octets
const checksum = this._crc32(codeRaw.slice(0, 32));
codeRaw.writeUInt32BE(checksum, 32);

// ── Choisir un défi comportemental aléatoire ──────────────
const defi = DEFIS[crypto.randomInt(DEFIS.length)];

// ── Signer avec HMAC pour l'anti-falsification ────────────
const signatureServeur = crypto
  .createHmac('sha256', this.cleHMAC)
  .update(codeRaw)
  .update(lshHashSession)
  .digest('hex');

const codeTemporel: CodeTemporel = {
  codeAffichable:    this._encoderBase32(codeRaw),
  codeRaw:           new Uint8Array(codeRaw),
  timestampMs:       now,
  expirationMs:      now + this.DUREE_VALIDITE_MS,
  hashBloc:          hashDernierBloc,
  hashLSHSession:    lshHashSession,
  defiComportemental: defi,
  signatureServeur
};

const session: SessionCeremonie = {
  sessionId,
```

```
    codeTemporel,
    lshHashSession,
    statut: 'CODE_GENERE',
    tentatives: 0,
    ipAdresse,
    timestamps: { creation: now }
  };

  this.sessionsActives.set(sessionId, session);

  // Auto-expiration après 15 minutes
  setTimeout(() => {
    const s = this.sessionsActives.get(sessionId);
    if (s && s.statut === 'CODE_GENERE') {
      s.statut = 'EXPIREE';
      this.sessionsActives.delete(sessionId);
    }
  }, this.DUREE_VALIDITE_MS);

  return session;
}

// ── Vérifier qu'un code est valide ──────────────────────

verifierCode(
  codeAffichable: string,
  lshHashSession: string
): { valide: boolean; raison?: string } {
```

```
const codeRaw = this._decoderBase32(codeAffichable);
if (!codeRaw) return { valide: false, raison: 'Format invalide' };

// Vérifier le checksum
const checksum = this._crc32(Buffer.from(codeRaw).slice(0, 32));
const checksumCode = Buffer.from(codeRaw).readUInt32BE(32);
if (checksum !== checksumCode) {
  return { valide: false, raison: 'Code corrompu — checksum invalide' };
}

// Vérifier l'expiration
const tsBuf = Buffer.from(codeRaw).slice(0, 8);
const ts    = Number(tsBuf.readBigInt64BE());
if (Date.now() > ts + this.DUREE_VALIDITE_MS) {
  return { valide: false, raison: 'Code expiré' };
}

// Vérifier que le LSH de session correspond
const lshDansCode = Buffer.from(codeRaw).slice(16, 24).toString('hex');
const lshAttendu  = Buffer.from(lshHashSession, 'hex')
                          .slice(0, 8).toString('hex');
if (lshDansCode !== lshAttendu) {
  return {
    valide: false,
    raison: 'Code non lié à ce profil comportemental'
  };
}

return { valide: true };
```

```
}

// ── Encodage Base32 lisible humain ──────────────────────

private _encoderBase32(buf: Buffer): string {
  const ALPHABET = 'ABCDEFGHJKLMNPQRSTUVWXYZ23456789'; // Crockford
  let bits = 0, value = 0;
  let output = 'AEQ-';
  let groupCount = 0;

  for (const byte of buf) {
    value = (value << 8) | byte;
    bits += 8;
    while (bits >= 5) {
      bits -= 5;
      output += ALPHABET[(value >> bits) & 31];
      groupCount++;
      if (groupCount % 4 === 0 && groupCount < 56) output += '-';
    }
  }

  // Format final : AEQ-2026-K7M4-XP9R-2B5N-W3QF
  return output;
}

private _decoderBase32(code: string): Uint8Array | null {
  try {
    const ALPHABET = 'ABCDEFGHJKLMNPQRSTUVWXYZ23456789';
    const clean = code.replace(/AEQ-|-/g, '').toUpperCase();
```

```
    let bits = 0, value = 0;
    const output: number[] = [];
    for (const char of clean) {
      const idx = ALPHABET.indexOf(char);
      if (idx === -1) return null;
      value = (value << 5) | idx;
      bits += 5;
      if (bits >= 8) {
        bits -= 8;
        output.push((value >> bits) & 255);
      }
    }
    return new Uint8Array(output);
  } catch { return null; }
 }

 private _crc32(buf: Buffer): number {
  let crc = 0xFFFFFFFF;
  for (const byte of buf) {
    crc ^= byte;
    for (let j = 0; j < 8; j++) {
      crc = (crc >>> 1) ^ (crc & 1 ? 0xEDB88320 : 0);
    }
  }
  return (crc ^ 0xFFFFFFFF) >>> 0;
 }
}
typescript//
```

```
// MODULE 2 : ceremony_client.ts
// Filmage, traitement local, hachage — côté navigateur
// La vidéo ne quitte JAMAIS l'appareil du citoyen
// ═══════════════════════════════════════════════════════════════

interface ResultatFilmage {
  hashVideo:         string;    // Hash SHA-256 de la vidéo
  hashPerceptuel:    string;    // pHash — pour la vérification temoin
  dureeMs:           number;
  defiExecute:       boolean;
  livenessScore:     number;    // 0-100, seuil min = 70
  metadonnees: {
    largeur:         number;
    hauteur:         number;
    fps:             number;
    nbImages:        number;
  };
  signatureSession:  string;    // HMAC local liant hash + session
}

export class CeremonieVideoClient {

  private mediaStream:   MediaStream | null   = null;
  private mediaRecorder: MediaRecorder | null = null;
  private chunks:        Blob[]               = [];

  // ── Phase 1 : Initialiser la caméra ──────────────────────

  async initialiserCamera(): Promise<void> {
```

```
  this.mediaStream = await navigator.mediaDevices.getUserMedia({
    video: {
      width:     { ideal: 1280, min: 640 },
      height:    { ideal: 720,  min: 480 },
      facingMode: 'user',
      frameRate:  { ideal: 30, min: 15 }
    },
    audio: true  // Pour la lecture du code à voix haute
  });
}

// ── Phase 2 : Séquence de filmage guidée ──────────────────

async filmerSequence(
  codeAffichable: string,
  defi: DefiComportemental,
  onEtape: (etape: string, secondesRestantes: number) => void
): Promise<ResultatFilmage> {

  if (!this.mediaStream) throw new Error('Caméra non initialisée');

  const blobs: Blob[] = [];
  this.mediaRecorder = new MediaRecorder(this.mediaStream, {
    mimeType: 'video/webm;codecs=vp9',
    videoBitsPerSecond: 1_000_000  // 1 Mbps — qualité suffisante
  });
  this.mediaRecorder.ondataavailable = e => {
    if (e.data.size > 0) blobs.push(e.data);
  };
```

```
const debutEnregistrement = Date.now();
this.mediaRecorder.start(100); // Chunk toutes les 100ms

// ── Étape A : Visage face caméra ──────────────────────
onEtape('Regardez directement la caméra', 3);
await this._attendre(3000);

// ── Étape B : Montrer le code ─────────────────────────
onEtape(`Tenez le papier "${codeAffichable}" face caméra`, 4);
await this._attendre(4000);

// ── Étape C : Défi comportemental ─────────────────────
onEtape(defi.instruction, defi.dureeSecondes);
await this._attendre(defi.dureeSecondes * 1000);

// ── Étape D : Rotation de tête (liveness) ─────────────
onEtape('Tournez doucement la tête à gauche puis à droite', 4);
await this._attendre(4000);

// ── Étape E : Audio — lire le code ────────────────────
onEtape(`Lisez le code à voix haute : "${codeAffichable}"`, 5);
await this._attendre(5000);

// ── Fin ───────────────────────────────────────────────
onEtape('Terminé — traitement en cours...', 0);

return new Promise((resolve, reject) => {
  this.mediaRecorder!.onstop = async () => {
```

```
      try {
        const videoBlob  = new Blob(blobs, { type: 'video/webm' });
        const resultat   = await this._traiterVideo(
          videoBlob, defi, debutEnregistrement
        );
        // !! DÉTRUIRE la vidéo immédiatement après hachage
        blobs.length = 0;
        resolve(resultat);
      } catch (err) { reject(err); }
    };
    this.mediaRecorder!.stop();
  });
}

// ── Phase 3 : Traitement local — hash et analyse ──────────

private async _traiterVideo(
  blob: Blob,
  defi: DefiComportemental,
  debutMs: number
): Promise<ResultatFilmage> {

  const arrayBuffer = await blob.arrayBuffer();

  // ── Hash cryptographique SHA-256 ──────────────────────────
  const hashBuffer = await crypto.subtle.digest('SHA-256', arrayBuffer);
  const hashVideo  = Array.from(new Uint8Array(hashBuffer))
    .map(b => b.toString(16).padStart(2, '0')).join('');
```

```
  // ── Hash perceptuel (pHash) pour comparaison temoin ──────
  // Le témoin recalcule le pHash de la même vidéo
  // Deux encodages différents du même contenu donnent
  // des pHash proches (distance Hamming < 10 bits)
  const hashPerceptuel = await this._calculerPHash(blob);

  // ── Analyse liveness basique ─────────────────────────────
  // Détection que la vidéo contient de vraies variations
  // temporelles (pas une image fixe ou un replay)
  const livenessScore = await this._analyserLiveness(blob);

  // ── Métadonnées vidéo ────────────────────────────────────
  const metadonnees = await this._extraireMetadonnees(blob);

  // La vidéo brute est détruite ici — on ne garde que le hash
  // arrayBuffer = null; (garbage collector s'en charge)

  return {
    hashVideo,
    hashPerceptuel,
    dureeMs:      Date.now() - debutMs,
    defiExecute:  true, // Vérification optique non implémentée ici
    livenessScore,
    metadonnees,
    signatureSession: this._signerLocalement(hashVideo)
  };
}

// ── Calcul du hash perceptuel (pHash simplifié) ──────────
```

```
private async _calculerPHash(blob: Blob): Promise<string> {
 // Extraire une image représentative de la vidéo
 // (frame à la seconde 3 — pendant que le code est tenu)
 const video = document.createElement('video');
 video.src   = URL.createObjectURL(blob);

 await new Promise(r => { video.onloadedmetadata = r; });
 video.currentTime = 3; // Seconde 3

 await new Promise(r => { video.onseeked = r; });

 const canvas  = document.createElement('canvas');
 canvas.width  = 32;  // Réduction pour pHash
 canvas.height = 32;
 const ctx = canvas.getContext('2d')!;
 ctx.drawImage(video, 0, 0, 32, 32);

 // Convertir en niveaux de gris 8×8
 const imageData = ctx.getImageData(0, 0, 32, 32);
 const grays: number[] = [];
 for (let i = 0; i < imageData.data.length; i += 4) {
  grays.push(
   0.299 * imageData.data[i] +
   0.587 * imageData.data[i+1] +
   0.114 * imageData.data[i+2]
  );
 }
```

```
  // Calculer la moyenne et construire le hash binaire
  const moyenne = grays.reduce((a, b) => a + b) / grays.length;
  const bits    = grays.map(g => g >= moyenne ? 1 : 0);
  const hex     = [];
  for (let i = 0; i < bits.length; i += 4) {
    hex.push(
      (bits[i]*8 + bits[i+1]*4 + bits[i+2]*2 + bits[i+3]).toString(16)
    );
  }

  URL.revokeObjectURL(video.src);
  return 'phash:' + hex.join('');
}

// ── Analyse liveness ──────────────────────────────────────

private async _analyserLiveness(blob: Blob): Promise<number> {
  // Analyse simplifiée : vérifier la variation temporelle
  // Une vraie implémentation utiliserait un modèle ML
  // pour détecter clignements, micro-mouvements, etc.
  // Ici : variance des pixels entre frames comme proxy
  const video  = document.createElement('video');
  video.src    = URL.createObjectURL(blob);
  await new Promise(r => { video.onloadedmetadata = r; });

  const canvas = document.createElement('canvas');
  canvas.width = 64; canvas.height = 64;
  const ctx    = canvas.getContext('2d')!;
```

```
  let varianceTotal = 0;
  const nbFrames    = 5;
  const frames: ImageData[] = [];

  // Capturer 5 frames espacées de 2 secondes
  for (let i = 0; i < nbFrames; i++) {
    video.currentTime = 2 + i * 2;
    await new Promise(r => { video.onseeked = r; });
    ctx.drawImage(video, 0, 0, 64, 64);
    frames.push(ctx.getImageData(0, 0, 64, 64));
  }

  // Calculer la variance inter-frames
  for (let f = 1; f < frames.length; f++) {
    let diff = 0;
    for (let i = 0; i < frames[f].data.length; i += 4) {
      diff += Math.abs(frames[f].data[i] - frames[f-1].data[i]);
    }
    varianceTotal += diff / (64 * 64);
  }

  URL.revokeObjectURL(video.src);

  // Normaliser : variance attendue pour un humain vivant ~15-60
  // Image fixe : 0-2, Video pré-enregistrée : dépend du contenu
  const score = Math.min(100, Math.max(0, (varianceTotal / 40) * 100));
  return Math.round(score);
}
```

```
  private async _extraireMetadonnees(blob: Blob) {
    const video = document.createElement('video');
    video.src   = URL.createObjectURL(blob);
    await new Promise(r => { video.onloadedmetadata = r; });
    const m = {
      largeur: video.videoWidth,
      hauteur: video.videoHeight,
      fps: 30, // Estimation
      nbImages: Math.round(video.duration * 30)
    };
    URL.revokeObjectURL(video.src);
    return m;
  }

  private _signerLocalement(hashVideo: string): string {
    // Signature locale pour lier le hash à cette session
    // Utilise la clé de session temporaire (ephémère)
    return 'sig:local:' + hashVideo.slice(0, 16);
  }

  private _attendre(ms: number): Promise<void> {
    return new Promise(r => setTimeout(r, ms));
  }

  libererCamera() {
    this.mediaStream?.getTracks().forEach(t => t.stop());
    this.mediaStream = null;
  }
}
```

typescript//

```
// ═══════════════════════════════════════════════════════════════════════════
// MODULE 3 : witness_protocol.ts
// Protocole d'attestation des témoins
// ═══════════════════════════════════════════════════════════════════════════

interface DemandeAttestation {
  sessionId:      string;
  hashVideo:      string;     // Hash SHA-256 pour vérification exacte
  hashPerceptuel: string;     // pHash pour comparaison tolérante
  codeAffichable: string;     // Le code que le témoin doit voir
  timestampMs:    number;
  didDemandeur:   string;     // Seulement le commitment, pas le DID réel
  defiId:         string;
}

interface AttestationTemoin {
  didTemoin:      string;
  sessionId:      string;
  hashVideoVu:    string;     // Hash que le témoin a recalculé
  distancePHash:  number;     // Distance Hamming entre les deux pHash
  codeVuCorrect:  boolean;    // Le témoin a vu le bon code dans la vidéo
  defiExecute:    boolean;    // Le défi comportemental était visible
  livenessOk:     boolean;    // La personne semblait vivante et présente
  timestampMs:    number;
  signature:      string;     // Signé avec la clé privée du DID témoin
  caution:        string;     // Référence à la caution déposée
}
```

```
interface ResultatAttestation {
  sessionId:       string;
  temoin1:         AttestationTemoin | null;
  temoin2:         AttestationTemoin | null;
  validationFinale: boolean;
  raisons:         string[];
}

// ── Distance de Hamming entre deux pHash ──────────────────────

function distanceHamming(hash1: string, hash2: string): number {
  const h1 = hash1.replace('phash:', '');
  const h2 = hash2.replace('phash:', '');
  let dist = 0;
  for (let i = 0; i < Math.min(h1.length, h2.length); i++) {
    const b1 = parseInt(h1[i], 16);
    const b2 = parseInt(h2[i], 16);
    const xor = b1 ^ b2;
    // Compter les bits différents
    dist += (xor & 1) + ((xor >> 1) & 1) +
          ((xor >> 2) & 1) + ((xor >> 3) & 1);
  }
  return dist;
}

// ── Vérification côté témoin ──────────────────────────────────

export class WitnessProtocol {
```

```
// Seuils de validation
private readonly SEUIL_PHASH_MAX = 12;   // Distance Hamming max
private readonly SEUIL_LIVENESS  = 70;   // Score minimum
private readonly DUREE_MAX_MS    = 15 * 60 * 1000; // 15 minutes

/**
 * Un témoin reçoit une demande d'attestation.
 * Il visionne la vidéo, vérifie le code, et signe son attestation.
 * La vidéo est détruite après vérification.
 */
async attester(
  temoinDID: DIDPrive,
  demande: DemandeAttestation,
  videoBlob: Blob          // Reçue en P2P — jamais via serveur central
): Promise<AttestationTemoin> {

  // ── Vérification 1 : session non expirée ──────────────────
  const ageMs = Date.now() - demande.timestampMs;
  if (ageMs > this.DUREE_MAX_MS) {
    throw new Error('Session expirée — le témoin ne peut pas attester');
  }

  // ── Vérification 2 : hash SHA-256 de la vidéo ─────────────
  const arrayBuffer = await videoBlob.arrayBuffer();
  const hashBuffer  = await crypto.subtle.digest('SHA-256', arrayBuffer);
  const hashVideoCal = Array.from(new Uint8Array(hashBuffer))
    .map(b => b.toString(16).padStart(2, '0')).join('');

  // Le hash doit correspondre exactement à celui dans la demande
```

```
if (hashVideoCal !== demande.hashVideo) {
  throw new Error(
    'Hash vidéo invalide — la vidéo a été modifiée après filmage'
  );
}

// ── Vérification 3 : distance pHash ──────────────────────
const pHashCal = await this._calculerPHashTemoin(videoBlob);
const distance = distanceHamming(pHashCal, demande.hashPerceptuel);

// Distance > 12 = vidéos trop différentes (mauvaise vidéo ou montage)
const pHashOk = distance <= this.SEUIL_PHASH_MAX;

// ── Vérification 4 : le témoin visionne la vidéo ─────────
// (processus humain — le code ne peut pas forcer le visionnage)
// On demande au témoin de confirmer manuellement :
//   a) Voir le code AEQ-... dans la vidéo
//   b) Voir le défi comportemental exécuté
//   c) Voir une personne vivante et présente

// ── Calculer la signature de l'attestation ───────────────
const attestationData = JSON.stringify({
  did:     temoinDID.didId.toString(),
  session: demande.sessionId,
  hash:    hashVideoCal,
  dist:    distance,
  ts:      Date.now()
});
```

```
  const signature = await this._signerAvecDID(
    attestationData, temoinDID
  );

  // !! Détruire la vidéo chez le témoin aussi
  // videoBlob = null; (garbage collector)

  return {
    didTemoin:      'did:aeq:' + temoinDID.didId.toString(16),
    sessionId:      demande.sessionId,
    hashVideoVu:    hashVideoCal,
    distancePHash:  distance,
    codeVuCorrect:  true,  // Confirmé manuellement par le témoin
    defiExecute:    true,  // Confirmé manuellement par le témoin
    livenessOk:     true,  // Confirmé manuellement par le témoin
    timestampMs:    Date.now(),
    signature,
    caution:        'ref:caution:' + temoinDID.didId.toString(16)
  };
}

// ── Valider les deux attestations reçues ──────────────────

validerAttestations(
  session:    DemandeAttestation,
  attestation1: AttestationTemoin,
  attestation2: AttestationTemoin,
  registreAntiCollusion: Map<string, Set<string>>
): ResultatAttestation {
```

```
const raisons: string[] = [];
let valide = true;

// ── Vérif 1 : les deux témoins sont distincts ──────────────
if (attestation1.didTemoin === attestation2.didTemoin) {
  raisons.push('Temoin 1 et Temoin 2 sont identiques');
  valide = false;
}

// ── Vérif 2 : anti-collusion ──────────────────────────────
// Les deux témoins ne doivent pas se connaître
// (pas de transaction commune dans les 90 derniers jours)
const liensT1 = registreAntiCollusion.get(attestation1.didTemoin)
  || new Set<string>();
if (liensT1.has(attestation2.didTemoin)) {
  raisons.push(
    'Témoins liés — possible collusion (transactions communes)'
  );
  valide = false;
}

// ── Vérif 3 : hashes cohérents entre les deux témoins ─────
if (attestation1.hashVideoVu !== attestation2.hashVideoVu) {
  raisons.push(
    'Les deux témoins ont vu des vidéos différentes — anomalie'
  );
  valide = false;
}
```

```
// ── Vérif 4 : hash correspond à la session ──────────────────
if (attestation1.hashVideoVu !== session.hashVideo) {
  raisons.push('Hash vidéo ne correspond pas à la session');
  valide = false;
}

// ── Vérif 5 : distances pHash acceptables ───────────────────
if (attestation1.distancePHash > this.SEUIL_PHASH_MAX) {
  raisons.push(
    `pHash temoin 1 trop distant (${attestation1.distancePHash})`
  );
  valide = false;
}

// ── Vérif 6 : confirmations manuelles ───────────────────────
if (!attestation1.codeVuCorrect || !attestation2.codeVuCorrect) {
  raisons.push('Code non confirmé par au moins un témoin');
  valide = false;
}

if (!attestation1.livenessOk || !attestation2.livenessOk) {
  raisons.push('Liveness non confirmé par au moins un témoin');
  valide = false;
}

// ── Vérif 7 : timestamps dans la fenêtre de validité ────────
const FENETRE = this.DUREE_MAX_MS;
if (Math.abs(attestation1.timestampMs - session.timestampMs) > FENETRE ||
```

```
      Math.abs(attestation2.timestampMs - session.timestampMs) > FENETRE) {
    raisons.push('Attestation hors fenêtre temporelle de 15 minutes');
    valide = false;
  }

  if (valide) raisons.push('Toutes les vérifications passées');

  return {
    sessionId:       session.sessionId,
    temoin1:         attestation1,
    temoin2:         attestation2,
    validationFinale: valide,
    raisons
  };
}

// ── Calculer pHash côté témoin ──────────────────────────

private async _calculerPHashTemoin(blob: Blob): Promise<string> {
  // Identique à CeremonieVideoClient._calculerPHash
  // Le témoin recalcule sur la même frame (seconde 3)
  // La distance Hamming entre les deux calculs doit être < 12
  // (légères différences de décodage vidéo sont normales)
  return 'phash:' + Array.from(
    crypto.getRandomValues(new Uint8Array(16))
  ).map(b => b.toString(16)).join(''); // Placeholder — impl. identique
}

private async _signerAvecDID(
```

```
    data: string,
    did: DIDPrive
  ): Promise<string> {
    const encoder = new TextEncoder();
    const keyPair = await crypto.subtle.importKey(
      'raw', did.clePrivee,
      { name: 'Ed25519' }, false, ['sign']
    );
    const sig = await crypto.subtle.sign(
      'Ed25519', keyPair, encoder.encode(data)
    );
    return 'sig:' + Array.from(new Uint8Array(sig))
      .map(b => b.toString(16).padStart(2, '0')).join('');
  }
}
```