

BonDriver Integrated Implementation Plan

自動チャンネルスキャン + DB保存 + ネットワークプロキシ 統合実装計画

概要

既存の [docs/BonDriverAutoScanPlan.md](#) と [docs/BonDriverProxyPlan.md](#) の2つの計画を統合し、以下の全プロジェクトを書き換える計画です。

統合プロジェクト一覧

1. **既存プロジェクト:** [b25-sys](#), [recisdb-rs](#)
2. **新規プロジェクト:** [recisdb-protocol](#), [recisdb-proxy](#), [bondriver-proxy-client](#)

1. 新規プロジェクト構成

```
recisdb-rs/                                # ワークスペースルート
  └── Cargo.toml                            # ワークスペース設定
  └── .gitignore

  └── b25-sys/                               # 既存 : ARIB-B25デコード
    └── 変更なし

  └── recisdb-rs/                            # 既存 : メインCLI
    ├── Cargo.toml                          # 拡張 : 依存関係追加
    ├── src/
    │   ├── main.rs                         # 拡張 : 新コマンド追加
    │   ├── context.rs                      # 拡張 : 新コマンド定義
    │   ├── channels.rs                     # 拡張 : DB-backed ChannelType追加
    │   ├── tuner/
    │   ├── commands/                       # 拡張 : scan/show/query追加
    │   └── database/
    │       ├── mod.rs
    │       ├── models.rs
    │       ├── bon_driver.rs
    │       ├── channel.rs
    │       └── scan_history.rs
    └── src/ts_analyzer/                     # **新規** : TS解析モジュール
      └── mod.rs
    └── src/ts_extractor/                   # **新規** : TS抽出モジュール
      └── mod.rs

  └── recisdb-protocol/                    # **新規** : プロトコル定義
    ├── Cargo.toml
    └── src/
```

```

    └── lib.rs
    └── types.rs          # メッセージ型、ChannelInfo、ChannelSelector
    └── codec.rs          # エンコード/デコード
    └── broadcast_region.rs # NID→放送種別・地域判定
    └── error.rs

    └── recisdb-proxy      # **新規**：サーバー
        ├── Cargo.toml
        └── recisdb-proxy.toml.example
    └── src/
        └── main.rs
        └── server/
            ├── listener.rs
            └── session.rs
        └── tuner/
            ├── pool.rs
            ├── shared.rs
            ├── channel_key.rs
            ├── lock.rs          # 排他/共有ロック管理
            ├── selector.rs       # チューナー自動選択（フォールバック付き）
            └── passive_scanner.rs # パッシブスキャン（配信中更新）
        └── scheduler/
            └── scan_scheduler.rs # 定期スキャンスケジューラー
        └── database/         # データベースモジュール
            ├── mod.rs
            ├── bon_driver.rs
            ├── channel.rs
            └── scan_history.rs

    └── bondriver-proxy-client/   # **新規**：クライアントDLL
        ├── Cargo.toml
        └── build.rs
    └── src/
        └── lib.rs
        └── bondriver/
            ├── interface.rs
            └── exports.rs
        └── client/
            ├── connection.rs
            └── buffer.rs

```

2. 統合アーキテクチャ設計

2.1 データベースの統合

AutoScan計画のデータベース設計を基に、Proxy計画でも共用する。

利用ファイル: [recisdb-protocol/src/types.rs](#)

```

/// チャンネル情報 (共用型)
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct ChannelInfo {
    // 一意識別キー (NID-SID-TSID-manual_sheet)
    pub nid: u16,                                // Network ID (SDTから取得)
    pub sid: u16,                                // Service ID
    pub tsid: u16,                               // Transport Stream ID
    pub manual_sheet: Option<u16>,             // ユーザー定義枝番 (None=標準)

    // チャンネル情報
    pub raw_name: Option<String>,           // 生のサービス名 (ARIB文字列)
    pub channel_name: Option<String>,         // 正規化されたチャンネル名
    pub physical_ch: Option<u8>,              // 物理チャンネル番号 (NITから取得)
    pub remote_control_key: Option<u8>,        // リモコンキーID (NITから取得)
    pub service_type: Option<u8>,              // サービス種別 (0x01=TV, 0x02=Radio)
    pub network_name: Option<String>,          // ネットワーク名 (NITから取得)

    // BonDriver固有情報 (スキヤン時に記録)
    pub bon_space: Option<u32>,                // BonDriverのSpace番号
    pub bon_channel: Option<u32>,               // BonDriverのChannel番号
}

impl ChannelInfo {
    /// 一意識別キーを生成
    pub fn unique_key(&self) -> (u16, u16, u16, Option<u16>) {
        (self.nid, self.sid, self.tsid, self.manual_sheet)
    }

    /// NID-SID-TSIDのみで比較 (manual_sheetを無視)
    pub fn service_key(&self) -> (u16, u16, u16) {
        (self.nid, self.sid, self.tsid)
    }
}

```

モジュール構成:

- [recisdb-rs/src/database/](#) - ローカルDB操作 (CLI)
- [recisdb-proxy/src/database/](#) - サーバーDB操作 (Proxy)

2.2 ChannelType の統合

既存の `channels.rs` を拡張し、DB-backedチャンネルをサポート。

変更ファイル: [recisdb-rs/src/channels.rs](#)

```

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ChannelType {
    // 既存型
    Terrestrial(u8, TsFilter),
    Catv(u8, TsFilter),
    BS(u8, TsFilter),
}

```

```

CS(u8, TsFilter),
BonCh(u8),
BonChSpace(ChannelSpace),
Undefined,

// **新規**: DB-backed型
Db {
    nid: u16,
    sid: u16,
    tsid: u16,
    manual_sheet: Option<u16>,
},
}

```

機能拡張:

- `ChannelType::from_db_info()` - DB情報からChannelType生成
- `ChannelType::to_db_key()` - DBキー変換
- 既存の `ChannelType::BS()` と `ChannelType::Db` の相互変換

2.3 依存関係管理

競合リスク:

- 既存: `futures-util`, `nom`, `clap`
- AutoScan追加: `rusqlite`, `serde`, `prettytable-rs`, `bitstream-io`
- Proxy追加: `tokio-rustls`, `rustls`, `tokio` (フルasync)

解決策: ワークスペースレベルで依存を統合管理

追加依存関係 (`recisdb-rs/Cargo.toml`):

```

# データベース関連
rusqlite = { version = "0.31", features = ["bundled"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
prettytable-rs = "0.10"
bitstream-io = "0.2"

# 非同期関連
tokio = { version = "1", features = ["full"] }
tokio-rustls = "0.25"
rustls = "0.22"

```

2.4 既存APIとの互換性

課題: 既存の `Tuner` enum と `AsyncInOutTriple` が、DB-backedチューニングとProxyを統合するか。

解決案:

2.4.1 Tuner Enum の拡張

```
// recisdb-rs/src/tuner/mod.rs
pub enum Tuner {
    #[cfg(target_os = "linux")]
    Character(character_device::Tuner),
    #[cfg(target_os = "linux")]
    DvbV5(dvbv5::Tuner),
    #[cfg(target_os = "windows")]
    BonDriver(windows::Tuner),

    // **新規**: DB-backedチューニング
    #[cfg(feature = "database")]
    Database(database::DatabaseTuner),
}
```

2.4.2 AsyncInOutTriple の拡張

```
// recisdb-rs/src/io.rs
impl<R, D, W> AsyncInOutTriple<R, D, W>
where
    R: AsyncRead + Unpin,
    D: StreamDecoder,
    W: AsyncWrite + Unpin,
{
    pub fn new_tune_with_db(
        device: &str,
        channel: &ChannelType,
        db_path: Option<&str>,
        decoder: Option<D>,
        output: W,
    ) -> Result<Self> {
        // DB-backed channel lookup
        if let ChannelType::Db { nid, sid, tsid, manual_sheet } = channel {
            // Database lookup and channel configuration
            // ...
        }
        // Fallback to original tuning logic
        // ...
    }
}
```

2.5 プラットフォーム差異の処理

AutoScan計画: TS解析が必要（実HW依存） **Proxy計画:** クロスプラットフォーム対応

統合案:

2.5.1 Feature-gated TS解析

```
# recisdb-rs/Cargo.toml
[features]
default = ["bg-runtime", "prioritized_card_reader"]
ts-analyzer = ["bitstream-io"] # TS解析モード
database = ["rusqlite", "serde", "prettytable-rs"] # データベースモード
```

2.5.2 プラットフォーム別実装

```
// recisdb-rs/src/tuner/mod.rs
impl Tuner {
    pub async fn open(path: &str) -> Result<Self, TunerError> {
        #[cfg(target_os = "linux")]
        {
            if path.contains("/dev/dvb/") {
                return Ok(Self::DvbV5(dvbv5::Tuner::new(path)?));
            }
            return Ok(Self::Character(character_device::Tuner::new(path)?));
        }
        #[cfg(target_os = "windows")]
        {
            return Ok(Self::BonDriver(windows::Tuner::new(path)?));
        }
    }
}
```

3. モジュール詳細設計

3.1 データベースモジュール (recisdb-rs/src/database/)

DBスキーマ (AutoScan計画から継承):

```
-- BonDriver管理
CREATE TABLE bon_drivers (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    dll_path TEXT UNIQUE NOT NULL,
    driver_name TEXT,
    version TEXT,
    -- スキャン設定 (チューナーごと)
    auto_scan_enabled INTEGER DEFAULT 1,          -- 自動スキャン有効/無効
    scan_interval_hours INTEGER DEFAULT 24,        -- スキャン間隔 (時間、0=無効)
    scan_priority INTEGER DEFAULT 0,               -- スキャン優先度 (高い順に実行)
    last_scan INTEGER,                            -- 最終スキャン時刻
    next_scan_at INTEGER,                         -- 次回スキャン予定時刻
    -- パッシブスキャン設定
```

```

passive_scan_enabled INTEGER DEFAULT 1, -- 配信中リアルタイム更新有効/無効
-- メタデータ
created_at INTEGER DEFAULT (strftime('%s', 'now')),
updated_at INTEGER DEFAULT (strftime('%s', 'now'))
);

-- チャンネル情報
CREATE TABLE channels (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    bon_driver_id INTEGER NOT NULL,
    -- 一意識別キー (NID-SID-TSID-manual_sheet)
    nid INTEGER NOT NULL, -- Network ID (SDTから取得)
    sid INTEGER NOT NULL, -- Service ID
    tsid INTEGER NOT NULL, -- Transport Stream ID
    manual_sheet INTEGER, -- ユーザー定義枝番 (NULL=標準)
    -- チャンネル情報
    raw_name TEXT, -- 生のサービス名 (ARIB文字列)
    channel_name TEXT, -- 正規化されたチャンネル名
    physical_ch INTEGER, -- 物理チャンネル番号 (NITから取得)
    remote_control_key INTEGER, -- リモコンキーID (NITから取得)
    service_type INTEGER, -- サービス種別 (0x01=TV, 0x02=Radio, etc.)
    network_name TEXT, -- ネットワーク名 (NITから取得)
    -- BonDriver固有情報
    bon_space INTEGER, -- BonDriverのSpace番号
    bon_channel INTEGER, -- BonDriverのChannel番号
    -- 状態管理
    is_enabled INTEGER DEFAULT 1, -- 有効/無効フラグ
    scan_time INTEGER, -- 最終スキャン時刻
    last_seen INTEGER, -- 最終検出時刻 (自動更新用)
    failure_count INTEGER DEFAULT 0, -- 連続チューニング失敗回数
    -- 選択優先度
    priority INTEGER DEFAULT 0, -- チャンネル選択優先度 (論理指定時に使用)
    -- メタデータ
    created_at INTEGER DEFAULT (strftime('%s', 'now')),
    updated_at INTEGER DEFAULT (strftime('%s', 'now')),
    UNIQUE(bon_driver_id, nid, sid, tsid, manual_sheet),
    FOREIGN KEY(bon_driver_id) REFERENCES bon_drivers(id) ON DELETE CASCADE
);

-- スキャン履歴
CREATE TABLE scan_history (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    bon_driver_id INTEGER NOT NULL,
    scan_time INTEGER DEFAULT (strftime('%s', 'now')),
    channel_count INTEGER,
    success INTEGER,
    error_message TEXT,
    FOREIGN KEY(bon_driver_id) REFERENCES bon_drivers(id) ON DELETE CASCADE
);

-- インデックス
CREATE INDEX idx_channels_bon_driver ON channels(bon_driver_id);
CREATE INDEX idx_channels_nid_sid_tsid ON channels(nid, sid, tsid);

```

```
CREATE INDEX idx_channels_enabled ON channels(is_enabled);
CREATE INDEX idx_scan_history_bon_driver ON scan_history(bon_driver_id);
```

主要機能:

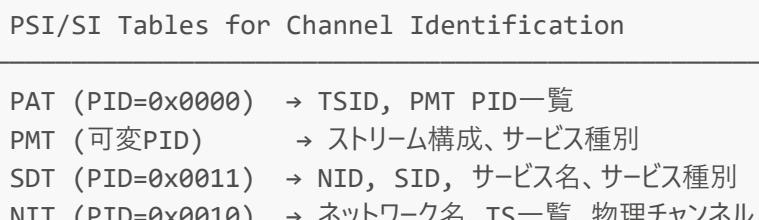
- `mod.rs` - データベース接続管理
- `models.rs` - データモデル定義
- `bon_driver.rs` - BonDriver CRUD操作
- `channel.rs` - チャンネル CRUD操作
- `scan_history.rs` - スキャン履歴管理

3.2 TS解析モジュール (recisdb-rs/src/ts_analyzer/)

主要機能:

- PATパケット解析 (Program Association Table) - TSID, SID一覧取得
- PMTパケット解析 (Program Map Table) - サービス種別検出
- **SDTパケット解析 (Service Description Table) - NID, サービス名取得**
- **NITパケット解析 (Network Information Table) - ネットワーク名、物理チャンネル情報取得**
- サービス種別検出 (TV/Radio/Data)

ARIB-STD-B10 テーブル構造:



変換ロジック:

```
// TSパケット → PAT/SDT/NIT → ChannelInfo
pub struct TsAnalyzer {
    pat: Option<PAT>,
    sdt: Option<SDT>,
    nit: Option<NIT>,
    pmt_map: HashMap<u16, PMT>, // program_number -> PMT
}

impl TsAnalyzer {
    /// TSパケットを投入し、テーブルを蓄積
    pub fn feed(&mut self, packet: &[u8; 188]) -> Result<(), TsError> {
        let pid = get_pid(packet);
        match pid {
            0x0000 => self.pat = Some(parse_pat(packet)?),
            _ => self.pmt_map.insert(pid, parse_pmt(packet)?),
            _ => self.sdt = Some(parse_sdt(packet)?),
            _ => self.nit = Some(parse_nit(packet)?),
        }
    }
}
```

```

0x0010 => self.nit = Some(parse_nit(packet)?),
0x0011 => self.sdt = Some(parse_sdt(packet)?),
_ => {
    // PMT PIDの場合
    if let Some(pat) = &self.pat {
        if let Some(pn) = pat.get_program_number_for_pid(pid) {
            self.pmt_map.insert(pn, parse_pmt(packet)?);
        }
    }
}
Ok(())
}

/// 解析完了判定 (PAT+SDT+NITが揃ったか)
pub fn is_complete(&self) -> bool {
    self.pat.is_some() && self.sdt.is_some() && self.nit.is_some()
}

/// チャンネル情報を抽出
pub fn extract_channels(&self) -> Vec<ChannelInfo> {
    let (sdt, nit) = match (&self.sdt, &self.nit) {
        (Some(sdt), Some(nit)) => (sdt, nit),
        _ => return Vec::new(), // 未完了なら空リスト
    };

    sdt.services.iter().map(|service| {
        ChannelInfo {
            nid: sdt.original_network_id, // SDTから取得
            sid: service.service_id,
            tsid: sdt.transport_stream_id,
            manual_sheet: None,
            raw_name: service.service_name.clone(),
            channel_name: normalize_channel_name(&service.service_name),
            physical_ch: nit.get_physical_channel(sdt.transport_stream_id),
            remote_control_key:
                nit.get_remote_control_key(service.service_id),
                service_type: Some(service.service_type),
                network_name: Some(nit.network_name.clone()),
                bon_space: None, // スキャン時に設定
                bon_channel: None, // スキャン時に設定
        }
    }).collect()
}
}
}

```

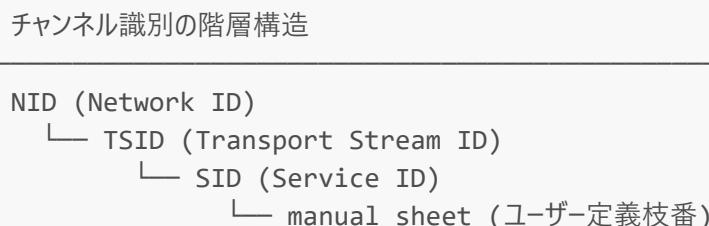
NID-SID-TSIDの取得元:

フィールド	取得元テーブル	備考
NID	SDT (original_network_id)	放送事業者識別
SID	SDT/PAT (service_id/program_number)	サービス識別

フィールド	取得元テーブル	備考
TSID	PAT/SDT (transport_stream_id)	トランスポートストリーム識別

3.3 チャンネルグループ化ロジック

一意識別キー: (NID, SID, TSID, manual_sheet)



グループ化ルール:

ケース	判定条件	動作
同一サービス	NID+SID+TSID一致	既存レコードを更新
異なるBonDriver	同一NID+SID+TSID、異なるbon_driver_id	各BonDriverで個別レコード作成
マニュアル分割	manual_sheet指定	同一NID+SID+TSIDでも別レコード
サービス消失	スキャン時に検出されない	is_enabled=0 に更新 (削除しない)

マニュアル枝番の用途:

```

/// manual_sheet は以下の用途で使用:
/// - 同一サービスを複数のプリセットに分けたい場合
/// - 時間帯別にチャンネルをグループ化したい場合
/// - ユーザーが明示的に別管理したい場合

// 例: BS朝日を複数プリセットで管理
// NID=4, SID=151, TSID=16400, manual_sheet=None → 標準
// NID=4, SID=151, TSID=16400, manual_sheet(Some(1)) → お気に入り1
// NID=4, SID=151, TSID=16400, manual_sheet(Some(2)) → お気に入り2
  
```

DBクエリ例:

```

-- 同一サービスの検索 (manual_sheetを無視)
SELECT * FROM channels
WHERE nid = ? AND sid = ? AND tsid = ?
ORDER BY manual_sheet NULLS FIRST;

-- 完全一致検索
  
```

```

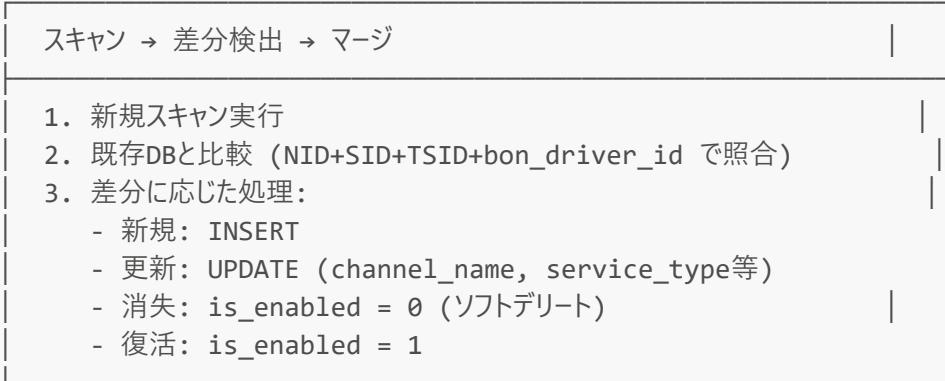
SELECT * FROM channels
WHERE nid = ? AND sid = ? AND tsid = ?
    AND (manual_sheet = ? OR (manual_sheet IS NULL AND ? IS NULL));

-- BonDriver別グループ化
SELECT bon_drivers.dll_path, COUNT(*) as channel_count
FROM channels
JOIN bon_drivers ON channels.bon_driver_id = bon_drivers.id
WHERE channels.is_enabled = 1
GROUP BY bon_drivers.id;

```

3.4 自動更新メカニズム

更新戦略:



実装:

```

// recisdb-rs/src/database/channel.rs
impl Database {
    /// スキャン結果をDBにマージ
    pub fn merge_scan_results(
        &self,
        bon_driver_id: i64,
        scanned_channels: &[ChannelInfo],
    ) -> Result<MergeResult> {
        let tx = self.conn.transaction()?;
        let mut result = MergeResult::default();

        // 既存チャンネルを取得
        let existing = self.get_channels_by_bon_driver(bon_driver_id)?;
        let existing_keys: HashSet<_> = existing.iter()
            .map(|c| (c.nid, c.sid, c.tsid))
            .collect();

        // スキャン結果を処理
        let scanned_keys: HashSet<_> = scanned_channels.iter()
            .map(|c| (c.nid, c.sid, c.tsid))

```

```

.collect();

for channel in scanned_channels {
    let key = (channel.nid, channel.sid, channel.tsid);
    if existing_keys.contains(&key) {
        // 更新
        self.update_channel(bon_driver_id, channel)?;
        result.updated += 1;
    } else {
        // 新規挿入
        self.insert_channel(bon_driver_id, channel)?;
        result.inserted += 1;
    }
}

// 消失したチャンネルを無効化
for existing_ch in &existing {
    let key = (existing_ch.nid, existing_ch.sid, existing_ch.tsid);
    if !scanned_keys.contains(&key) && existing_ch.is_enabled {
        self.disable_channel(existing_ch.id)?;
        result.disabled += 1;
    }
}

tx.commit()?;
Ok(result)
}

/// 定期スキャン用: 最終スキャンから指定時間経過したBonDriverを取得
pub fn get_stale_bon_drivers(&self, max_age_secs: i64) ->
Result<Vec<BonDriver>> {
    let threshold = chrono::Utc::now().timestamp() - max_age_secs;
    self.conn.prepare(
        "SELECT * FROM bon_drivers WHERE last_scan < ? OR last_scan IS NULL"
    )?.query_map([threshold], |row| /* ... */)
}
}

#[derive(Default)]
pub struct MergeResult {
    pub inserted: usize,
    pub updated: usize,
    pub disabled: usize,
}

```

自動更新トリガー:

トリガー	条件	動作
起動時スキャン	--auto-scan フラグ	起動時に全BonDriverをスキャン
定期スキャン	scan_interval 設定	指定間隔で自動スキャン

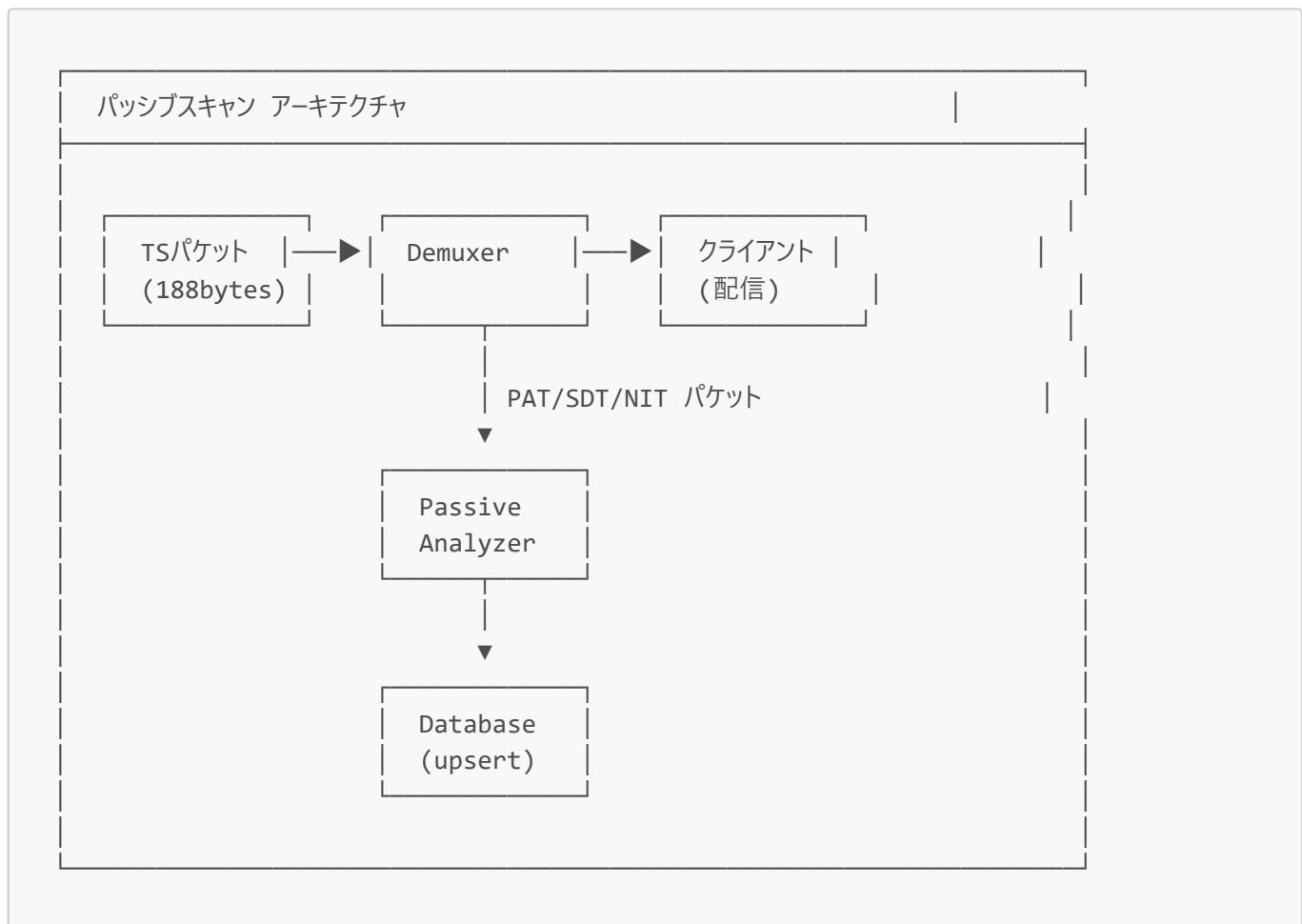
トリガー	条件	動作
手動スキャン	recisdb scan コマンド	ユーザー明示実行
チューニング失敗時	signal_level < threshold	該当チャンネルを再スキャン

設定ファイル拡張:

```
# ~/.config/recisdb/config.toml
[scan]
auto_scan_on_startup = true          # 起動時自動スキャン
timeout_per_channel_secs = 5         # チャンネルあたりのスキャンタイムアウト
retry_on_failure = 3                 # 失敗時リトライ回数
disable_after_failures = 5           # N回連続失敗でチャンネル無効化
# 注: scan_interval_hours はDB側 (bon_drivers.scan_interval_hours) で管理
```

3.5 パッシブスキャン (配信中リアルタイム更新)

概要: TS配信中は常にPAT/SDT/NITパケットが流れているため、これを監視することでチャンネル情報をリアルタイム更新できる。



実装:

```
// recisdb-proxy/src/tuner/pассив_scanner.rs

use tokio::sync::mpsc;
use crate::ts_analyzer::TsAnalyzer;
use crate::database::Database;

/// パッシブスキャナー: 配信中のTSからチャンネル情報を抽出・更新
pub struct PassiveScanner {
    analyzer: TsAnalyzer,
    db: Arc<Database>,
    bon_driver_id: i64,
    current_channel: Option<(u32, u32)>, // (space, channel)
    last_update: Instant,
    update_interval: Duration, // 更新間隔 (デフォルト: 30秒)
}

impl PassiveScanner {
    pub fn new(db: Arc<Database>, bon_driver_id: i64) -> Self {
        Self {
            analyzer: TsAnalyzer::new(),
            db,
            bon_driver_id,
            current_channel: None,
            last_update: Instant::now(),
            update_interval: Duration::from_secs(30),
        }
    }

    /// TS/パケットを投入 (配信パイプラインから呼び出し)
    pub fn feed(&mut self, packet: &[u8; 188]) {
        let pid = get_pid(packet);

        // PSI/SIパケットのみ処理
        match pid {
            0x0000 | 0x0010 | 0x0011 => {
                if let Err(e) = self.analyzer.feed(packet) {
                    debug!("Passive scan feed error: {}", e);
                }
            }
            _ => return,
        }

        // 定期的にDB更新
        if self.last_update.elapsed() >= self.update_interval {
            self.try_update_db();
        }
    }

    /// 解析結果をDBに反映
    fn try_update_db(&mut self) {
        if !self.analyzer.is_complete() {
            return;
        }
    }
}
```

```

let channels = self.analyzer.extract_channels();
if channels.is_empty() {
    return;
}

// BonDriver固有情報を追加
let channels: Vec<_> = channels.into_iter().map(|mut ch| {
    if let Some((space, channel)) = self.current_channel {
        ch.bon_space = Some(space);
        ch.bon_channel = Some(channel);
    }
    ch
}).collect();

// 差分更新（軽量版：last_seenのみ更新 or 変更があればフル更新）
match self.db.passive_update_channels(self.bon_driver_id, &channels) {
    Ok(updated) => {
        if updated > 0 {
            info!("Passive scan: updated {} channels", updated);
        }
    }
    Err(e) => {
        warn!("Passive scan DB update failed: {}", e);
    }
}

self.last_update = Instant::now();
self.analyzer.reset(); // 次のサイクル用にリセット
}

/// チャンネル変更時に呼び出し
pub fn on_channel_changed(&mut self, space: u32, channel: u32) {
    self.current_channel = Some((space, channel));
    self.analyzer.reset();
}
}
}

```

DB操作（軽量更新）：

```

// recisdb-rs/src/database/channel.rs
impl Database {
    /// パッシブスキャン用：last_seen更新 + 変更検出時のみフル更新
    pub fn passive_update_channels(
        &self,
        bon_driver_id: i64,
        channels: &[ChannelInfo],
    ) -> Result<usize> {
        let now = chrono::Utc::now().timestamp();
        let mut updated = 0;

        for channel in channels {

```

```

let existing = self.get_channel_by_key(
    bon_driver_id,
    channel.nid,
    channel.sid,
    channel.tsid,
)?;

match existing {
    Some(existing) => {
        // last_seen を更新
        self.conn.execute(
            "UPDATE channels SET last_seen = ?, failure_count = 0
WHERE id = ?",
            [now, existing.id],
        );
    }

    // チャンネル名やservice_typeに変更があればフル更新
    if existing.channel_name != channel.channel_name
        || existing.service_type != channel.service_type
    {
        self.update_channel(bon_driver_id, channel)?;
        updated += 1;
    }
}

None => {
    // 新規チャンネル発見
    self.insert_channel(bon_driver_id, channel)?;
    updated += 1;
    info!("Passive scan: new channel discovered: NID={}, SID={}, TSID={}",
        channel.nid, channel.sid, channel.tsid);
}
}

Ok(updated)
}
}

```

SharedTunerへの統合:

```

// recisdb-proxy/src/tuner/shared.rs
pub struct SharedTuner {
    // ... 既存フィールド
    passive_scanner: Option<Mutex<PassiveScanner>>,
}

impl SharedTuner {
    /// TSデータを配信（パッシブスキャン付き）
    pub async fn broadcast_with_passive_scan(&self, packet: &[u8; 188]) {
        // クライアントへ配信
        let _ = self.tx.send(packet.to_vec());
    }
}

```

```
// パッシブスキャン（有効な場合のみ）
if let Some(scanner) = &self.passive_scanner {
    scanner.lock().await.feed(packet);
}
}
```

3.6 チューナーごとのスキャンスケジューリング

DB設定による個別制御:

```
-- チューナーごとのスキャン設定例
UPDATE bon_drivers SET
    auto_scan_enabled = 1,          -- 自動スキャン有効
    scan_interval_hours = 12,       -- 12時間ごと
    scan_priority = 10,            -- 高優先度
    passive_scan_enabled = 1       -- パッシブスキャン有効
WHERE dll_path = 'BonDriver_PT3-S.dll';

-- 地デジチューナーは週1回スキャン
UPDATE bon_drivers SET
    auto_scan_enabled = 1,
    scan_interval_hours = 168,     -- 168時間 = 1週間
    scan_priority = 0,             -- 低優先度
    passive_scan_enabled = 1
WHERE dll_path = 'BonDriver_PT3-T.dll';

-- 特定チューナーは手動スキャンのみ
UPDATE bon_drivers SET
    auto_scan_enabled = 0,         -- 自動スキャン無効
    passive_scan_enabled = 0       -- パッシブスキャンも無効
WHERE dll_path = 'BonDriver_OLD.dll';
```

スケジューラー実装:

```
// recisdb-proxy/src/scheduler/scan_scheduler.rs

pub struct ScanScheduler {
    db: Arc<Database>,
    tuner_pool: Arc<TunerPool>,
}

impl ScanScheduler {
    /// スケジューラーのメインループ
    pub async fn run(&self) {
        let mut interval = tokio::time::interval(Duration::from_secs(60));

        loop {
```

```
interval.tick().await;

// スキャン対象のBonDriverを取得
let due_drivers = match self.db.get_due_bon_drivers() {
    Ok(drivers) => drivers,
    Err(e) => {
        error!("Failed to get due bon_drivers: {}", e);
        continue;
    }
};

// 優先度順にソート
let mut due_drivers = due_drivers;
due_drivers.sort_by(|a, b| b.scan_priority.cmp(&a.scan_priority));

for driver in due_drivers {
    info!("Starting scheduled scan for: {}", driver.dll_path);

    match self.scan_bon_driver(&driver).await {
        Ok(result) => {
            // 次回スキャン時刻を更新
            let next_scan = chrono::Utc::now().timestamp()
                + (driver.scan_interval_hours as i64 * 3600);
            self.db.update_next_scan(driver.id, next_scan).ok();

            info!("Scheduled scan complete: {} (inserted={}, updated= {},
{}), disabled={})", driver.dll_path,
                    result.merge_result.inserted,
                    result.merge_result.updated,
                    result.merge_result.disabled);
        }
        Err(e) => {
            warn!("Scheduled scan failed for {}: {}", driver.dll_path,
e);
        }
    }
}

// Database側
impl Database {
    /// スキャン期限が来ているBonDriverを取得
    pub fn get_due_bon_drivers(&self) -> Result<Vec<BonDriver>> {
        let now = chrono::Utc::now().timestamp();
        self.conn.prepare(
            "SELECT * FROM bon_drivers
             WHERE auto_scan_enabled = 1
               AND scan_interval_hours > 0
               AND (next_scan_at IS NULL OR next_scan_at <= ?)
             ORDER BY scan_priority DESC, next_scan_at ASC"
        )?.query_map([now], |row| /* ... */)
    }
}
```

```
    }

    /// 次回スキャン時刻を更新
    pub fn update_next_scan(&self, bon_driver_id: i64, next_scan_at: i64) ->
Result<()> {
    self.conn.execute(
        "UPDATE bon_drivers SET next_scan_at = ?, last_scan = strftime('%s',
'now')
        WHERE id = ?",
        [next_scan_at, bon_driver_id],
    )?;
    Ok(())
}
}
```

CLIでのスキャン設定管理

```
# スキヤン設定の表示  
recisdb scan-config --device BonDriver_PT3-S.dll  
  
# スキヤン間隔の変更  
recisdb scan-config --device BonDriver_PT3-S.dll --interval 12  
  
# 自動スキンを無効化  
recisdb scan-config --device BonDriver_PT3-S.dll --auto-scan off  
  
# パッシブスキンを無効化  
recisdb scan-config --device BonDriver_PT3-S.dll --passive-scan off  
  
# 優先度の変更  
recisdb scan-config --device BonDriver_PT3-S.dll --priority 10  
  
# 即時スキン実行（スケジュール無視）  
recisdb scan --device BonDriver_PT3-S.dll --force
```

コマンド定義の追加

```
// recisdb-rs/src/context.rs に追加
#[clap(name = "scan-config")]
ScanConfig {
    /// BonDriver DLL path
    #[clap(short, long)]
    device: String,
    /// Database file path
    #[clap(long)]
    database: Option<String>,
    /// Set scan interval in hours (0 = disable auto scan)
}
```

```

#[clap(long)]
interval: Option<u32>,

/// Enable/disable auto scan (on/off)
#[clap(long)]
auto_scan: Option<String>,

/// Enable/disable passive scan (on/off)
#[clap(long)]
passive_scan: Option<String>,

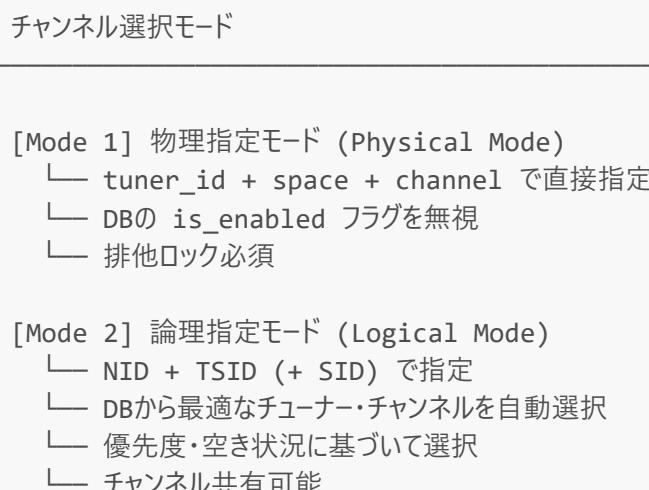
/// Set scan priority (higher = scanned first)
#[clap(long)]
priority: Option<i32>,
}

```

3.7 チャンネル選択ロジックと排他ロック

3.7.1 チャンネル選択モード

2種類のチャンネル指定方式:



選択モード判定:

```

// recisdb-protocol/src/types.rs
#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum ChannelSelector {
    /// 物理指定: DBの有効/無効を無視して直接選局
    Physical {
        tuner_id: String,           // BonDriver識別子
        space: u32,                 // チューナー空間
        channel: u32,               // 物理チャンネル番号
    }
}

```

```

    },
    /// 論理指定: NID/TSIDでDB検索→最適チューナー選択
    Logical {
        nid: u16,           // Network ID
        tsid: u16,          // Transport Stream ID
        sid: Option<u16>,   // Service ID (任意、フィルタ用)
    },
}

impl ChannelSelector {
    /// 物理指定かどうか
    pub fn is_physical(&self) -> bool {
        matches!(self, Self::Physical { .. })
    }

    /// DBのis_enabledを確認すべきか
    pub fn should_check_enabled(&self) -> bool {
        !self.is_physical() // 論理指定のみチェック
    }
}

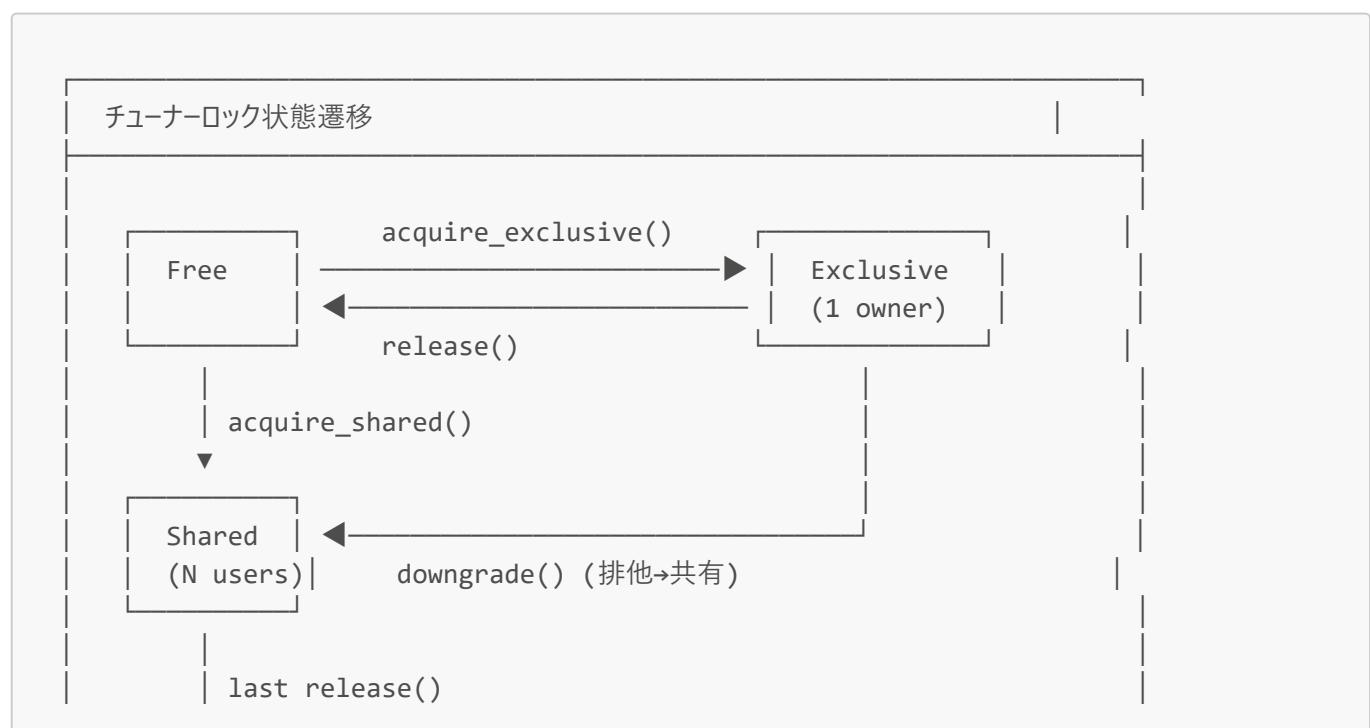
```

3.7.2 排他ロック機構

ロックの種類:

ロック種別	用途	競合
Exclusive (排他)	物理指定でのチャンネル変更	全ロックと競合
Shared (共有)	論理指定での同一チャンネル視聴	Exclusiveのみ競合

ロック状態遷移:



▼
Free

実装:

```
// recisdb-proxy/src/tuner/lock.rs

use std::sync::Arc;
use tokio::sync::{RwLock, Semaphore, OwnedSemaphorePermit};

/// チューナーロック管理
pub struct TunerLock {
    /// 排他/共有ロック用セマフォ
    /// - 排他: 全permitを取得
    /// - 共有: 1 permitのみ取得
    semaphore: Arc<Semaphore>,
    max_permits: u32,

    /// 現在のチャンネル（共有判定用）
    current_channel: RwLock<Option<ChannelKey>>,

    /// 共有クライアント数
    shared_count: AtomicU32,
}

impl TunerLock {
    const MAX_SHARED_CLIENTS: u32 = 100;

    pub fn new() -> Self {
        Self {
            semaphore: Arc::new(Semaphore::new(Self::MAX_SHARED_CLIENTS as
usize)),
            max_permits: Self::MAX_SHARED_CLIENTS,
            current_channel: RwLock::new(None),
            shared_count: AtomicU32::new(0),
        }
    }

    /// 排他ロック取得（物理指定用）
    /// - 全クライアントが解放されるまで待機
    /// - チャンネル変更が可能
    pub async fn acquire_exclusive(&self) -> Result<ExclusiveLockGuard, LockError> {
        // 全permitを取得（排他）
        let permits = self.semaphore
            .clone()
            .acquire_many_owned(self.max_permits)
            .await
    }
}
```

```
.map_err(|_| LockError::Closed)?;

Ok(ExclusiveLockGuard {
    _permits: permits,
    lock: self,
})
}

/// 共有ロック取得（論理指定用）
/// - 同一チャンネルなら即座に取得可能
/// - 異なるチャンネルならエラー（別チューナーを探す）
pub async fn acquire_shared(
    &self,
    channel: &ChannelKey,
) -> Result<SharedLockGuard, LockError> {
    // 現在のチャンネルを確認
    let current = self.current_channel.read().await;

    match &*current {
        Some(current_ch) if current_ch == channel => {
            // 同一チャンネル: 共有OK
            drop(current);
        }
        Some(_) => {
            // 異なるチャンネル: このチューナーは使用不可
            return Err(LockError::ChannelMismatch);
        }
        None => {
            // 未使用: 排他ロックで初期化が必要
            return Err(LockError::NotInitialized);
        }
    }
}

// 1 permitを取得（共有）
let permit = self.semaphore
    .clone()
    .acquire_owned()
    .await
    .map_err(|_| LockError::Closed)?;

self.shared_count.fetch_add(1, Ordering::SeqCst);

Ok(SharedLockGuard {
    _permit: permit,
    lock: self,
})
}

/// 排他ロック中にチャンネルを設定
pub async fn set_channel(&self, channel: ChannelKey) {
    let mut current = self.current_channel.write().await;
    *current = Some(channel);
}
```

```
/// 排他→共有ヘダウングレード
pub async fn downgrade(
    guard: ExclusiveLockGuard<'_>,
    channel: &ChannelKey,
) -> SharedLockGuard<'_> {
    // チャンネルを設定
    guard.lock.set_channel(channel.clone()).await;

    // 排他ロックを解放し、共有ロックとして1 permitのみ保持
    let semaphore = guard.lock.semaphore.clone();
    let permits_to_release = guard.lock.max_permits - 1;

    // ガードをドロップせずにpermitを部分解放
    // (実装の詳細は省略)

    guard.lock.shared_count.fetch_add(1, Ordering::SeqCst);

    SharedLockGuard {
        _permit: /* 1 permit */,
        lock: guard.lock,
    }
}

pub struct ExclusiveLockGuard<'a> {
    _permits: OwnedSemaphorePermit,
    lock: &'a TunerLock,
}

pub struct SharedLockGuard<'a> {
    _permit: OwnedSemaphorePermit,
    lock: &'a TunerLock,
}

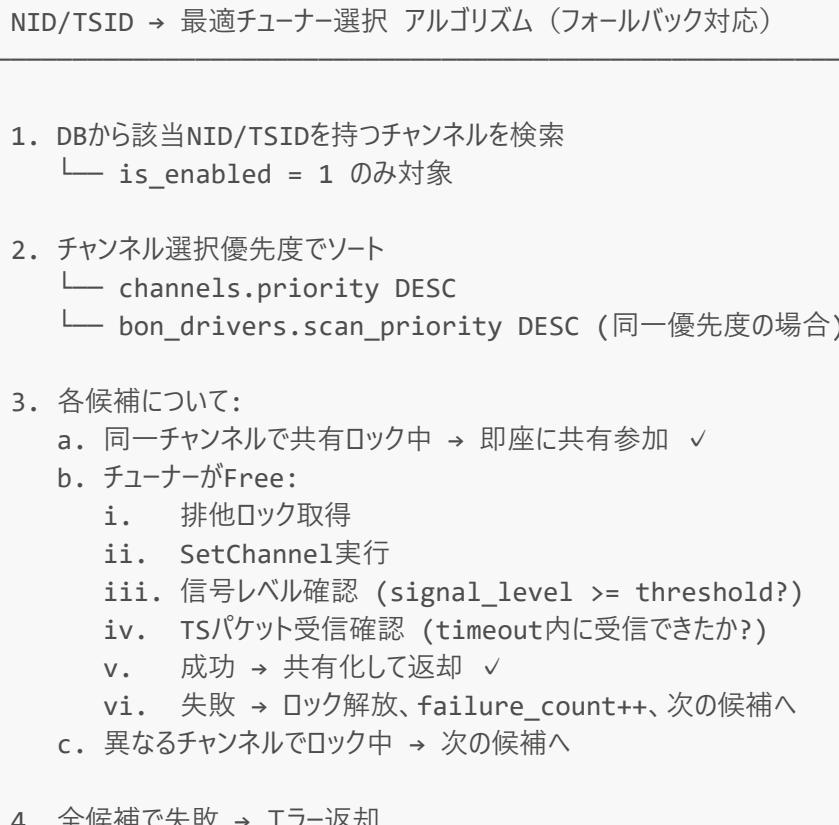
impl Drop for SharedLockGuard<'_> {
    fn drop(&mut self) {
        let count = self.lock.shared_count.fetch_sub(1, Ordering::SeqCst);
        if count == 1 {
            // 最後の共有クライアント：チャンネルをクリア
            // (非同期コンテキスト外なのでspawnで処理)
        }
    }
}

#[derive(Debug, thiserror::Error)]
pub enum LockError {
    #[error("Tuner is locked on different channel")]
    ChannelMismatch,
    #[error("Tuner not initialized (needs exclusive lock first)")]
    NotInitialized,
    #[error("Lock system closed")]
    Closed,
    #[error("Lock timeout")]
}
```

```
    Timeout,
}
```

3.7.3 論理指定でのチューナー自動選択

選択アルゴリズム（フォールバック付き）：



フォールバック条件:

条件	動作	DB更新
ロック取得失敗 (別チャンネル使用中)	次の候補へ	なし
SetChannel失敗	ロック解放、次の候補へ	failure_count++
信号レベル < threshold	ロック解放、次の候補へ	failure_count++
TSパケット受信タイムアウト	ロック解放、次の候補へ	failure_count++
failure_count >= max_failures	候補から除外	is_enabled=0

インデックス追加 (channelsテーブル - priorityは3.1のスキーマで定義済み):

```
-- 優先度付きインデックス（論理指定での高速検索用）
CREATE INDEX idx_channels_nid_tsid_priority
ON channels(nid, tsid, priority DESC, is_enabled);
```

実装:

```
// recisdb-proxy/src/tuner(selector.rs)

use crate::database::Database;
use crate::tuner::{TunerPool, TunerLock, ChannelKey};

const SIGNAL_THRESHOLD: f64 = 5.0;
const TUNE_TIMEOUT_MS: u64 = 3000;
const TS_RECEIVE_TIMEOUT_MS: u64 = 2000;
const MAX_FAILURE_COUNT: i32 = 5;

pub struct TunerSelector {
    db: Arc<Database>,
    tuner_pool: Arc<TunerPool>,
}

impl TunerSelector {
    /// 論理指定でチューナーを選択・ロック取得（フォールバック対応）
    pub async fn select_by_logical(
        &self,
        nid: u16,
        tsid: u16,
        sid: Option<u16>,
    ) -> Result<SelectedTuner, SelectError> {
        // 1. DBから候補チャンネルを優先度順に取得
        let candidates = self.db.get_channels_by_nid_tsid_ordered(nid, tsid, sid)?;

        if candidates.is_empty() {
            return Err(SelectError::ChannelNotFound { nid, tsid });
        }

        let mut last_error = None;

        // 2. 各候補を試行（フォールバック）
        for candidate in &candidates {
            let tuner_id = &candidate.bon_driver_path;
            let channel_key = ChannelKey {
                tuner_id: tuner_id.clone(),
                space: candidate.bon_space.unwrap_or(0),
                channel: candidate.bon_channel.unwrap_or(0),
            };
            let tuner = match self.tuner_pool.get_tuner(tuner_id).await {
                Some(t) => t,
                None => continue,
            };

            if tuner.is_locked() {
                continue;
            }

            if tuner.priority() >= SIGNAL_THRESHOLD {
                break;
            }

            if tuner.sid() == sid {
                break;
            }

            if tuner.tsid() == tsid {
                break;
            }

            if tuner.nid() == nid {
                break;
            }
        }

        if last_error.is_none() {
            return Err(SelectError::NoCandidate);
        }

        let tuner = last_error.unwrap();
        let lock = TunerLock::new(tuner);
        let channel = lock.get_channel();
        let space = lock.get_space();
        let key = lock.get_key();
        let id = lock.get_id();

        Ok(SelectedTuner {
            tuner,
            channel,
            space,
            key,
            id,
        })
    }
}
```

```

    None => {
        debug!("Tuner not found: {}, trying next", tuner_id);
        continue;
    }
};

// 3a. 同一チャンネルで共有中なら参加（既にTS受信中なので検証不要）
if let Ok(guard) = tuner.lock.acquire_shared(&channel_key).await {
    info!("Joined existing shared tuner: {} for NID={}, TSID={}",
          tuner_id, nid, tsid);
    return Ok(SelectedTuner {
        tuner,
        guard: LockGuard::Shared(guard),
        channel_info: candidate.clone(),
    });
}

// 3b. Freeなら排他ロック→チャンネル設定→検証→共有化
if let Ok(exclusive) = tuner.lock.try_acquire_exclusive().await {
    // チャンネル設定を試行
    match self.try_tune_and_verify(
        &tuner,
        candidate.bon_space.unwrap_or(0),
        candidate.bon_channel.unwrap_or(0),
    ).await {
        Ok(_) => {
            // 成功: 共有ロックへダウングレード
            let shared = TunerLock::downgrade(exclusive,
&channel_key).await;

            // failure_countをリセット
            self.db.reset_failure_count(candidate.id).ok();

            info!("Successfully tuned: {} for NID={}, TSID={}",
                  tuner_id, nid, tsid);

            return Ok(SelectedTuner {
                tuner,
                guard: LockGuard::Shared(shared),
                channel_info: candidate.clone(),
            });
        }
        Err(e) => {
            // 失敗: ロックは自動解放 (exclusive drop)
            warn!("Tune failed for {} (NID={}, TSID={}): {}, trying
next",
                  tuner_id, nid, tsid, e);

            // failure_countをインクリメント
            let new_count =
self.db.increment_failure_count(candidate.id)?;

            // 閾値を超えたたらチャンネルを無効化
            if new_count >= MAX_FAILURE_COUNT {

```

```

        warn!("Disabling channel {} due to {} consecutive
failures",
            candidate.id, new_count);
        self.db.disable_channel(candidate.id)?;
    }

    last_error = Some(e);
    continue; // 次の候補へフォールバック
}
}

// 3c. ロック取得失敗（別チャンネル使用中）、次の候補へ
debug!("Tuner {} is busy with different channel, trying next",
tuner_id);
}

// 4. 全候補で失敗
Err(last_error.map(SelectError::TuneFailed)
    .unwrap_or(SelectError::AllTunersBusy))
}

/// チューニング実行+信号・TS受信検証
async fn try_tune_and_verify(
    &self,
    tuner: &SharedTuner,
    space: u32,
    channel: u32,
) -> Result<(), TuneError> {
    // Step 1: SetChannel
    tuner.set_channel(space, channel).await
        .map_err(|e| TuneError::SetChannelFailed(e.to_string()))?;

    // Step 2: 信号ロック待機
    let lock_start = Instant::now();
    let lock_timeout = Duration::from_millis(TUNE_TIMEOUT_MS);

    loop {
        if lock_start.elapsed() > lock_timeout {
            return Err(TuneError::SignalLockTimeout);
        }

        let signal = tuner.get_signal_level().await;
        if signal >= SIGNAL_THRESHOLD {
            break; // 信号ロック成功
        }

        tokio::time::sleep(Duration::from_millis(100)).await;
    }

    // Step 3: TSパケット受信確認
    let ts_start = Instant::now();
    let ts_timeout = Duration::from_millis(TS_RECEIVE_TIMEOUT_MS);
}

```

```

loop {
    if ts_start.elapsed() > ts_timeout {
        return Err(TuneError::TsReceiveTimeout);
    }

    if tuner.has_received_ts_packets().await {
        break; // TS受信確認
    }

    tokio::time::sleep(Duration::from_millis(50)).await;
}

Ok(())
}

#[derive(Debug, thiserror::Error)]
pub enum TuneError {
    #[error("SetChannel failed: {0}")]
    SetChannelFailed(String),
    #[error("Signal lock timeout (no signal or weak signal)")]
    SignalLockTimeout,
    #[error("TS packet receive timeout")]
    TsReceiveTimeout,
}

#[derive(Debug, thiserror::Error)]
pub enum SelectError {
    #[error("Channel not found: NID={nid}, TSID={tsid}")]
    ChannelNotFound { nid: u16, tsid: u16 },
    #[error("All tuners are busy")]
    AllTunersBusy,
    #[error("Tune failed: {0}")]
    TuneFailed(TuneError),
    #[error("Tuner not found: {0}")]
    TunerNotFound(String),
    #[error("Lock failed: {0}")]
    LockFailed(LockError),
}

// Database側
impl Database {
    /// failure_countをインクリメント
    pub fn increment_failure_count(&self, channel_id: i64) -> Result<i32> {
        self.conn.execute(
            "UPDATE channels SET failure_count = failure_count + 1 WHERE id = ?",
            [channel_id],
        )?;

        let count: i32 = self.conn.query_row(
            "SELECT failure_count FROM channels WHERE id = ?",
            [channel_id],
            |row| row.get(0),
        )?;
    }
}

```

```
        Ok(count)
    }

    /// failure_countをリセット
    pub fn reset_failure_count(&self, channel_id: i64) -> Result<()> {
        self.conn.execute(
            "UPDATE channels SET failure_count = 0, last_seen = strftime('%s',
            'now') WHERE id = ?",
            [channel_id],
        )?;
        Ok(())
    }
}

impl TunerSelector {
    /// 物理指定でチューナーを選択・排他ロック取得
    pub async fn select_by_physical(
        &self,
        tuner_id: &str,
        space: u32,
        channel: u32,
    ) -> Result<SelectedTuner, SelectError> {
        let tuner = self.tuner_pool.get_tuner(tuner_id).await
            .ok_or_else(|| SelectError::TunerNotFound(tuner_id.to_string()))?;

        // 排他ロック取得（待機）
        let exclusive = tuner.lock.acquire_exclusive().await
            .map_err(|e| SelectError::LockFailed(e))?;

        // チャンネル設定（DBの有効/無効を無視）
        tuner.set_channel(space, channel).await?;

        let channel_key = ChannelKey {
            tuner_id: tuner_id.to_string(),
            space,
            channel,
        };

        // DBからチャンネル情報を取得（あれば）
        let channel_info = self.db.get_channel_by_physical(tuner_id, space,
            channel)
            .ok()
            .flatten();

        // 共有ロックへダウングレード（他クライアントも参加可能に）
        let shared = TunerLock::downgrade(exclusive, &channel_key).await;

        Ok(SelectedTuner {
            tuner,
            guard: LockGuard::Shared(shared),
            channel_info,
        })
    }
}
```

```

}

// Database側
impl Database {
    /// NID/TSIDで優先度順にチャンネルを取得
    pub fn get_channels_by_nid_tsid_ordered(
        &self,
        nid: u16,
        tsid: u16,
        sid: Option<u16>,
    ) -> Result<Vec<ChannelWithDriver>> {
        let mut query = String::from(
            "SELECT c.*, bd.dll_path as bon_driver_path, bd.scan_priority
             FROM channels c
             JOIN bon_drivers bd ON c.bon_driver_id = bd.id
             WHERE c.nid = ? AND c.tsid = ? AND c.is_enabled = 1"
        );

        if sid.is_some() {
            query.push_str(" AND c.sid = ?");
        }

        query.push_str(" ORDER BY c.priority DESC, bd.scan_priority DESC");

        // ... 実行
    }
}

```

3.7.4 クライアント側のチャンネルリスト同期

プロトコル拡張:

```

// recisdb-protocol/src/types.rs に追加

/// チャンネルリスト同期用メッセージ
#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum ChannelListMessage {
    /// クライアント→サーバー: チャンネルリスト要求
    Request {
        /// フィルタ条件 (任意)
        filter: Option<ChannelFilter>,
    },
    /// サーバー→クライアント: チャンネルリスト応答
    Response {
        channels: Vec<ClientChannelInfo>,
        /// リスト生成時刻 (差分更新用)
        timestamp: i64,
    },
    /// サーバー→クライアント: 差分更新通知
}

```

```

Update {
    added: Vec<ClientChannelInfo>,
    updated: Vec<ClientChannelInfo>,
    removed: Vec<ChannelKey>,
    timestamp: i64,
},
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct ChannelFilter {
    pub nid: Option<u16>,
    pub tsid: Option<u16>,
    pub network_type: Option<NetworkType>, // Terrestrial, BS, CS
    pub enabled_only: bool,
}

/// クライアントに送信するチャンネル情報
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct ClientChannelInfo {
    // 識別子
    pub nid: u16,
    pub sid: u16,
    pub tsid: u16,

    // 表示用
    pub channel_name: String,
    pub network_name: Option<String>,
    pub service_type: u8,
    pub remote_control_key: Option<u8>,

    // BonDriver互換用 (TVTest表示用)
    pub space_name: String,           // "BS" / "CS" / "地デジ" など
    pub channel_display_name: String, // "BS朝日" など

    // 優先度
    pub priority: i32,
}

```

クライアント側実装 (bondriver-proxy-client):

```

// bondriver-proxy-client/src/client/channel_cache.rs

use std::collections::HashMap;

/// クライアント側チャンネルキャッシュ
pub struct ChannelCache {
    /// NID-TSID → チャンネル情報マップ
    channels: HashMap<(u16, u16), Vec<ClientChannelInfo>>,

    /// Space/Channel → NID-TSID マッピング (BonDriver互換用)
    physical_to_logical: HashMap<(u32, u32), (u16, u16, u16)>,
}

```

```

    /// 最終同期時刻
    last_sync: i64,
}

impl ChannelCache {
    /// サーバーからチャンネルリストを同期
    pub async fn sync(&mut self, client: &mut ProxyClient) -> Result<()> {
        let response = client.send(ChannelListMessage::Request {
            filter: Some(ChannelFilter {
                enabled_only: true,
                ..Default::default()
            }),
        }).await?;

        if let ChannelListMessage::Response { channels, timestamp } = response {
            self.channels.clear();
            self.physical_to_logical.clear();

            for ch in channels {
                // NID-TSID でグループ化
                self.channels
                    .entry((ch.nid, ch.tsid))
                    .or_default()
                    .push(ch.clone());
            }

            // Space/Channel マッピング構築 (BonDriver互換)
            // TVTestはSpace/Channelでアクセスするため
        }

        self.last_sync = timestamp;
    }

    Ok(())
}

/// NID-TSIDからチャンネル情報を取得
pub fn get_by_nid_tsid(&self, nid: u16, tsid: u16) -> Option<&[ClientChannelInfo]> {
    self.channels.get(&(nid, tsid)).map(|v| v.as_slice())
}

/// BonDriverのSpace/ChannelからNID-TSID-SIDを解決
pub fn resolve_physical(&self, space: u32, channel: u32) -> Option<(u16, u16, u16)> {
    self.physical_to_logical.get(&(space, channel)).copied()
}
}

```

NIDによる放送種別・地域判定:

```
// recisdb-protocol/src/broadcast_region.rs
```

```

/// 放送種別
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub enum BroadcastType {
    Terrestrial, // 地上波
    BS,          // BS
    CS,          // CS (CS1, CS2)
}

/// 放送地域 (地上波のみ)
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub enum TerrestrialRegion {
    Hokkaido,      // 北海道
    Tohoku,         // 東北
    Kanto,          // 関東広域
    Koshinetsu,     // 甲信越
    Hokuriku,       // 北陸
    Tokai,          // 東海
    Kinki,          // 近畿広域
    Chugoku,        // 中国
    Shikoku,        // 四国
    Kyushu,         // 九州
    Okinawa,        // 沖縄
    Unknown(u16),   // 不明なNID
}

/// NIDから放送種別・地域を判定
/// 参考: ARIB TR-B14, TR-B15
pub fn classify_nid(nid: u16) -> (BroadcastType, Option<TerrestrialRegion>) {
    match nid {
        // BS (NID = 4)
        4 => (BroadcastType::BS, None),

        // CS (NID = 6, 7, 10)
        6 | 7 | 10 => (BroadcastType::CS, None),

        // 地上波 (NID = 0x7FE0 ~ 0xFFFF: 地域別)
        // 上位4ビット = 0x7, 下位12ビットで地域識別
        0x7FE0..=0xFFFF => {
            let region = match nid {
                // 北海道 (複数NID)
                0x7FE0..=0x7FE7 => TerrestrialRegion::Hokkaido,
                // 関東広域圏
                0x7FE8 => TerrestrialRegion::Kanto,
                // 近畿広域圏
                0x7FE9 => TerrestrialRegion::Kinki,
                // 中京広域圏 (東海)
                0x7FEA => TerrestrialRegion::Tokai,
                // 岡山・香川
                0x7FEB => TerrestrialRegion::Chugoku,
                // 島根・鳥取
                0x7FEC => TerrestrialRegion::Chugoku,
                // 北海道 (追加)
                0x7FF0..=0x7FF7 => TerrestrialRegion::Hokkaido,
                // その他
            }
            (BroadcastType::Terrestrial, Some(region))
        }
    }
}

```

```

        _ => TerrestrialRegion::Unknown(nid),
    };
    (BroadcastType::Terrestrial, Some(region))
}

// 県域放送 (NID = 32721 ~ 32767: 0x7FD1 ~ 0xFFFF の一部)
// 詳細な地域マッピングは別途定義
_ => {
    // 不明なNIDは地上波として扱う
    if nid >= 0x7F00 {
        (BroadcastType::Terrestrial,
        Some(TerrestrialRegion::Unknown(nid)))
    } else {
        // 完全に不明
        (BroadcastType::Terrestrial,
        Some(TerrestrialRegion::Unknown(nid)))
    }
}
}

impl TerrestrialRegion {
    pub fn display_name(&self) -> &'static str {
        match self {
            Self::Hokkaido => "北海道",
            Self::Tohoku => "東北",
            Self::Kanto => "関東",
            Self::Koshinetsu => "甲信越",
            Self::Hokuriku => "北陸",
            Self::Tokai => "東海",
            Self::Kinki => "近畿",
            Self::Chugoku => "中国",
            Self::Shikoku => "四国",
            Self::Kyushu => "九州",
            Self::Okinawa => "沖縄",
            Self::Unknown(_) => "その他",
        }
    }
}
}

```

チューナー空間の自動生成:

```

// bondriver-proxy-client/src/client/space_builder.rs

use std::collections::{HashMap, BTreeMap};
use crate::broadcast_region::{BroadcastType, TerrestrialRegion, classify_nid};

/// 自動生成されたチューナー空間
#[derive(Debug, Clone)]
pub struct TuningSpace {
    pub space_id: u32,
    pub name: String,
}

```

```

pub broadcast_type: BroadcastType,
pub region: Option<TerrestrialRegion>,
pub channels: Vec<SpaceChannel>,
}

#[derive(Debug, Clone)]
pub struct SpaceChannel {
    pub index: u32,           // Space内のインデックス
    pub nid: u16,
    pub tsid: u16,
    pub sid: u16,
    pub display_name: String, // "NHK総合", "BS朝日" など
    pub remote_key: Option<u8>,
}

/// チャンネルリストからチューナー空間を自動生成
pub struct SpaceBuilder {
    /// 生成されたSpace一覧
    spaces: BTreeMap<u32, TuningSpace>,
    /// NID → Space ID マッピング
    nid_to_space: HashMap<u16, u32>,
    /// (space, channel) → (nid, tsid, sid) マッピング
    physical_to_logical: HashMap<(u32, u32), (u16, u16, u16)>,
    /// 次のSpace ID
    next_space_id: u32,
}

impl SpaceBuilder {
    pub fn new() -> Self {
        Self {
            spaces: BTreeMap::new(),
            nid_to_space: HashMap::new(),
            physical_to_logical: HashMap::new(),
            next_space_id: 0,
        }
    }

    /// チャンネルリストからSpace構造を構築
    pub fn build_from_channels(&mut self, channels: &[ClientChannelInfo]) {
        // 1. チャンネルをBroadcastType + Region でグループ化
        let mut groups: HashMap<(BroadcastType, Option<TerrestrialRegion>), Vec<&ClientChannelInfo>>
            = HashMap::new();

        for ch in channels {
            let (btype, region) = classify_nid(ch.nid);
            groups.entry((btype, region)).or_default().push(ch);
        }

        // 2. 固定順序でSpaceを生成: 地デジ各地域 → BS → CS
        let mut ordered_keys: Vec<_> = groups.keys().cloned().collect();
        ordered_keys.sort_by(|a, b| {
            // 地デジ → BS → CS の順
            let type_order = |t: &BroadcastType| match t {

```

```
        BroadcastType::Terrestrial => 0,
        BroadcastType::BS => 1,
        BroadcastType::CS => 2,
    };
    type_order(&a.0).cmp(&type_order(&b.0))
});

for key in ordered_keys {
    let (btype, region) = key;
    let chs = groups.get(&key).unwrap();

    // Space名を生成
    let space_name = match btype {
        BroadcastType::Terrestrial => {
            if let Some(r) = region {
                format!("地デジ ({})", r.display_name())
            } else {
                "地デジ".to_string()
            }
        }
        BroadcastType::BS => "BS".to_string(),
        BroadcastType::CS => "CS".to_string(),
    };

    let space_id = self.next_space_id;
    self.next_space_id += 1;

    // チャンネルをリモコンキー順 or SID順でソート
    let mut sorted_chs: Vec<_> = chs.iter().cloned().collect();
    sorted_chs.sort_by(|a, b| {
        // リモコンキーがあればそれで、なければSIDで
        match (a.remote_control_key, b.remote_control_key) {
            (Some(ka), Some(kb)) => ka.cmp(&kb),
            (Some(_), None) => std::cmp::Ordering::Less,
            (None, Some(_)) => std::cmp::Ordering::Greater,
            (None, None) => a.sid.cmp(&b.sid),
        }
    });
}

// SpaceChannelを生成
let space_channels: Vec<SpaceChannel> = sorted_chs
    .iter()
    .enumerate()
    .map(|(idx, ch)| {
        // マッピングを登録
        self.physical_to_logical.insert(
            (space_id, idx as u32),
            (ch.nid, ch.tsid, ch.sid),
        );
    });

SpaceChannel {
    index: idx as u32,
    nid: ch.nid,
    tsid: ch.tsid,
```

```
        sid: ch.sid,
        display_name: ch.channel_name.clone(),
        remote_key: ch.remote_control_key,
    }
})
.collect();

// NID → Space マッピング
for ch in &sorted_chs {
    self.nid_to_space.insert(ch.nid, space_id);
}

// Spaceを登録
self.spaces.insert(space_id, TuningSpace {
    space_id,
    name: space_name,
    broadcast_type: btype,
    region,
    channels: space_channels,
});
}

/// Space数を取得
pub fn num_spaces(&self) -> u32 {
    self.spaces.len() as u32
}

/// Space名を取得
pub fn get_space_name(&self, space: u32) -> Option<&str> {
    self.spaces.get(&space).map(|s| s.name.as_str())
}

/// Space内のチャンネル数を取得
pub fn num_channels_in_space(&self, space: u32) -> u32 {
    self.spaces.get(&space)
        .map(|s| s.channels.len() as u32)
        .unwrap_or(0)
}

/// チャンネル名を取得
pub fn get_channel_name(&self, space: u32, channel: u32) -> Option<&str> {
    self.spaces.get(&space)
        .and_then(|s| s.channels.get(channel as usize))
        .map(|c| c.display_name.as_str())
}

/// (Space, Channel) → (NID, TSID, SID) 解決
pub fn resolve(&self, space: u32, channel: u32) -> Option<(u16, u16, u16)> {
    self.physical_to_logical.get(&(space, channel)).copied()
}
}
```

TVTest互換: EnumChannelName実装:

```
// bondriver-proxy-client/src/bondriver/exports.rs

impl BonDriverProxy {
    /// チャンネルリスト受信時にSpace構造を自動構築
    fn on_channel_list_received(&mut self, channels: Vec<ClientChannelInfo>) {
        let mut builder = SpaceBuilder::new();
        builder.build_from_channels(&channels);
        self.space_builder = builder;
    }

    /// TVTest用: チューニング空間数
    pub fn enum_tuning_space(&self) -> u32 {
        self.space_builder.num_spaces()
    }

    /// TVTest用: チューニング空間名
    pub fn enum_tuning_space_name(&self, space: u32) -> Option<String> {
        self.space_builder.get_space_name(space).map(|s| s.to_string())
    }

    /// TVTest用: チャンネル数
    pub fn enum_channel(&self, space: u32) -> u32 {
        self.space_builder.num_channels_in_space(space)
    }

    /// TVTest用: チャンネル名列挙
    pub fn enum_channel_name(&self, space: u32, index: u32) -> Option<String> {
        self.space_builder.get_channel_name(space, index).map(|s| s.to_string())
    }

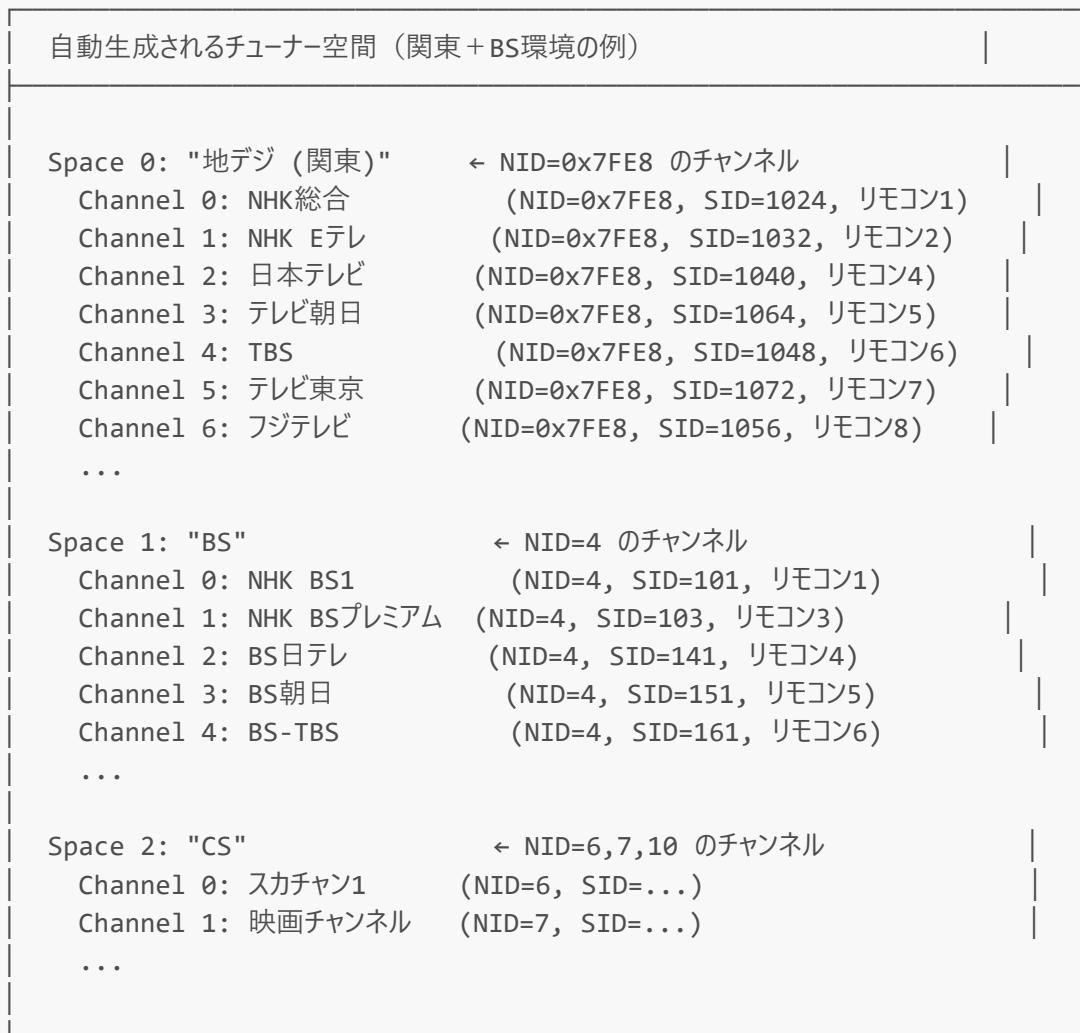
    /// TVTest用: チャンネル選択
    /// Space/Channel → NID/TSID/SID変換 → サーバーへ論理指定で要求
    pub fn set_channel(&mut self, space: u32, channel: u32) -> bool {
        // Space/ChannelからNID-TSID-SIDを解決
        if let Some((nid, tsid, sid)) = self.space_builder.resolve(space, channel) {
            // 論理指定でサーバーへ要求
            self.send_set_channel(ChannelSelector::Logical {
                nid,
                tsid,
                sid: Some(sid),
            })
        } else {
            // 解決失敗: 物理指定でフォールバック
            self.send_set_channel(ChannelSelector::Physical {
                tuner_id: self.current_tuner.clone(),
                space,
                channel,
            })
        }
    }
}
```

```

    }
}

```

生成されるSpace構造の例:



複数地域が混在する場合:

```

Space 0: "地デジ (関東)"      ← NID=0x7FE8
Space 1: "地デジ (近畿)"      ← NID=0x7FE9
Space 2: "BS"
Space 3: "CS"

```

3.7.5 全体シーケンス

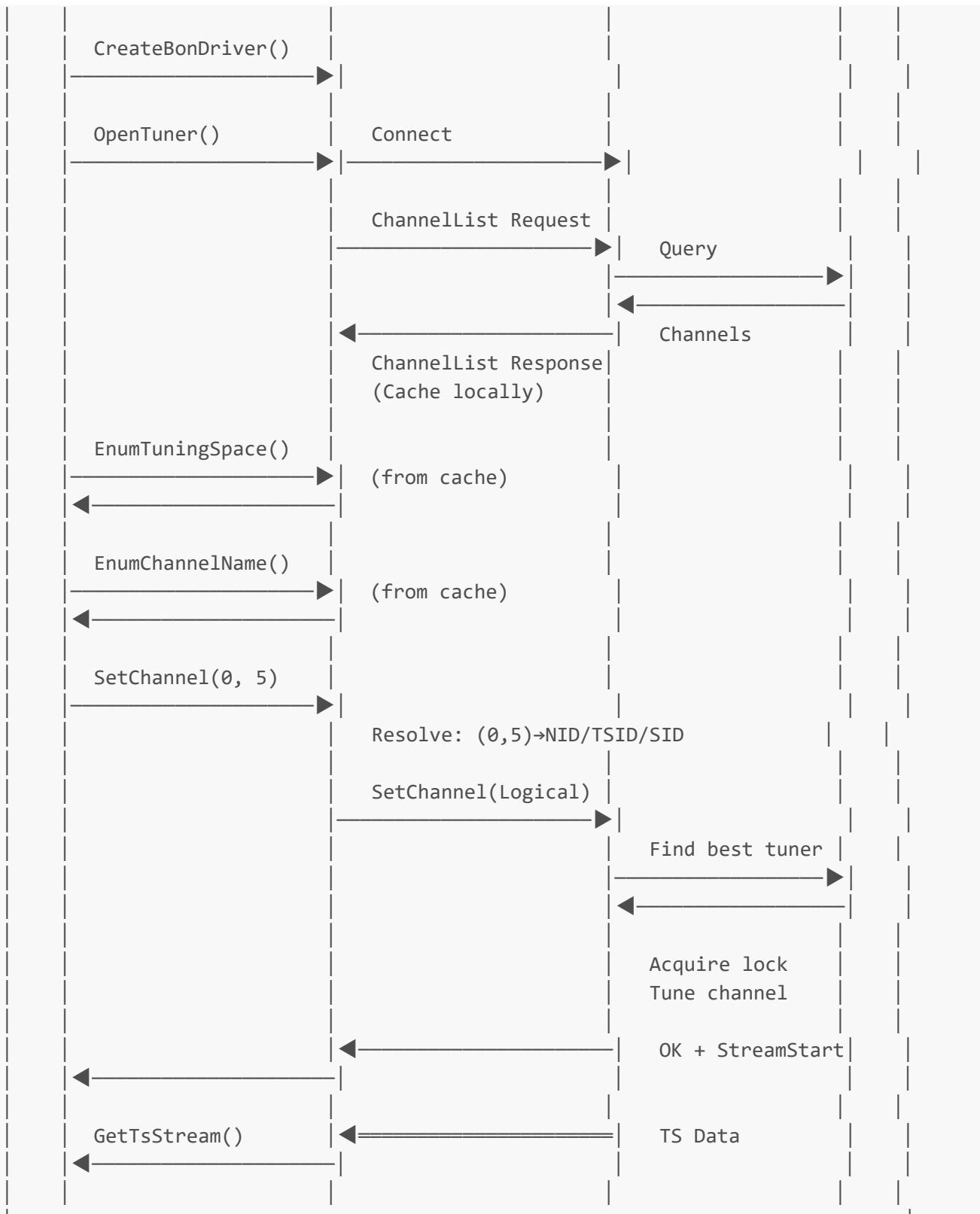
TVTest起動 → チャンネル選択 シーケンス

TVTest

BonDriver DLL

Proxy Server

Database



3.8 プロトコルモジュール (recisdb-protocol/)

Type	名称	方向	用途
0x0001	OpenTuner	C→S	チューナーオープン要求
0x0002	CloseTuner	C→S	チューナークローズ
0x0010	ChannelListRequest	C→S	チャンネルリスト要求

Type	名称	方向	用途
0x0011	ChannelListResponse	S→C	チャンネルリスト応答
0x0012	ChannelListUpdate	S→C	チャンネルリスト差分更新
0x0101	SetChannelPhysical	C→S	物理指定チャンネル設定
0x0102	SetChannelLogical	C→S	論理指定チャンネル設定 (NID/TSID)
0x0301	GetSignalLevel	C→S	信号レベル取得
0x0401	StartStream	C→S	ストリーム開始
0x0403	StreamData	S→C	TSデータ送信

フレームフォーマット:

```
// recisdb-protocol/src/codec.rs
#[derive(Debug, Clone)]
pub struct Frame {
    pub magic: [u8; 4],           // "BNDP"
    pub length: u32,              // LE
    pub message_type: u16,        // LE
    pub payload: Vec<u8>,
}
```

3.4 プロキシサーバー (recisdb-proxy/)

TunerPool設計:

```
// recisdb-proxy/src/tuner/pool.rs
pub struct TunerPool {
    tuners: RwLock<HashMap<ChannelKey, Arc<SharedTuner>>>,
}

impl TunerPool {
    pub async fn get_or_create(
        &self,
        tuner_path: &str,
        channel: &Channel,
    ) -> Result<Arc<SharedTuner>> {
        let key = ChannelKey::from(tuner_path, channel);

        // 既存チューナー再利用
        if let Some(tuner) = self.tuners.read().await.get(&key) {
            return Ok(Arc::clone(tuner));
        }

        // 新規作成
        let shared = self.create_tuner(tuner_path, channel).await?;
        self.tuners.write().await.insert(key, Arc::clone(&shared));
    }
}
```

```

        Ok(shared)
    }
}

```

セッション管理:

- TCP/TLSリスナー (tokio-rustls)
- クライアント認証 (証明書ベース)
- セッションごとのチャンネル共有

3.5 クライアントDLL (bondriver-proxy-client/)

IBonDriver vtable実装:

```

// bondriver-proxy-client/src/bondriver/interface.rs
#[repr(C)]
pub struct IBonDriver1 {
    pub vtable: *const IBonDriver1VTable,
    // ...
}

#[repr(C)]
pub struct IBonDriver1VTable {
    pub OpenTuner: unsafe extern "C" fn(*mut IBonDriver1) -> i32,
    pub CloseTuner: unsafe extern "C" fn(*mut IBonDriver1),
    pub SetChannel: unsafe extern "C" fn(*mut IBonDriver1, space: u32, channel: u32) -> i32,
    pub GetTsStream: unsafe extern "C" fn(
        *mut IBonDriver1,
        *mut u8,
        *mut u32,
        *mut u32,
    ) -> i32,
    // ... v2, v3
}

```

リングバッファ:

```

// bondriver-proxy-client/src/client/buffer.rs
const RING_BUFFER_SIZE: usize = 2 * 1024 * 1024; // 2MB

pub struct TsRingBuffer {
    buffer: Box<[u8; RING_BUFFER_SIZE]>,
    write_pos: AtomicUsize, // Receiver Task
    read_pos: AtomicUsize, // Main Thread (GetTsStream)
}

impl TsRingBuffer {
    pub fn write(&self, data: &[u8]) -> Result<usize> {

```

```
    let write = self.write_pos.load(Ordering::Acquire);
    let read = self.read_pos.load(Ordering::Acquire);
    // ロックフリー実装
    // ...
}
}
```

4. CLIコマンド拡張

4.1 新コマンド定義

変更ファイル: `recisdb-rs/src/context.rs`

```
#[derive(Parser, Debug)]
pub enum Commands {
    // 既存コマンド
    #[clap(name = "tune")]
    Tune { /* ... */ },
    #[clap(name = "decode")]
    Decode { /* ... */ },
    #[clap(name = "enumerate")]
    Enumerate { /* ... */ },
    #[clap(name = "checksignal")]
    CheckSignal { /* ... */ },
    // **新規コマンド**: チャンネルスキャン
    #[clap(name = "scan")]
    Scan {
        /// BonDriver DLL path
        #[clap(short, long)]
        device: String,
        /// Output database file path
        #[clap(short, long)]
        database: Option<String>,
        /// Recreate database from scratch
        #[clap(short, long)]
        recreate: bool,
        /// Timeout in seconds
        #[clap(short, long, default_value = "30")]
        timeout: u64,
    },
    // **新規コマンド**: チャンネル表示
    #[clap(name = "show")]
}
```

```

Show {
    /// BonDriver DLL path
#[clap(short, long)]
device: String,

    /// Database file path
#[clap(short, long)]
database: Option<String>,

    /// Output format (json/table)
#[clap(short, long, default_value = "table")]
format: String,
},

// **新規コマンド**: チャンネルクエリ
#[clap(name = "query")]
Query {
    /// BonDriver DLL path
#[clap(short, long)]
device: String,

    /// Database file path
#[clap(short, long)]
database: Option<String>,

    /// Channel to query (e.g., BS101, CS110_1)
#[clap(short, long)]
channel: Option<String>,

    /// NID to query
#[clap(short, long)]
nid: Option<u16>,

    /// SID to query
#[clap(short, long)]
sid: Option<u16>,

    /// TSID to query
#[clap(short, long)]
tsid: Option<u16>,

    /// Manual sheet number
#[clap(short, long)]
manual_sheet: Option<u16>,
},
}

```

4.2 コマンド実装

scanコマンド ([recisdb-rs/src/commands/scan.rs](#)):

スキャンフロー図:

BonDriver自動チャンネルスキャン フロー

1. BonDriver初期化
 - └─ OpenTuner() → EnumTuningSpace() → EnumChannelName()

2. Space/Channel全走査

```

for space in 0..num_spaces:
    for ch in 0..num_channels[space]:
        SetChannel(space, ch)
        wait_for_lock(timeout)
        if signal_level > threshold:
            collect_ts_packets(duration)
            analyze_pat_sdt_nit()
            store_channel_info()
    
```

3. DB更新

- └─ merge_scan_results() → 新規INSERT/既存UPDATE/消失無効化

4. 結果出力

- └─ inserted: N, updated: M, disabled: K

```

use std::time::{Duration, Instant};
use crate::database::Database;
use crate::ts_analyzer::TsAnalyzer;
use crate::tuner::windows::UnTunedTuner;

const TS_PACKET_SIZE: usize = 188;
const SIGNAL_THRESHOLD: f64 = 5.0;
const ANALYSIS_DURATION_MS: u64 = 2000;

pub async fn scan_bon_driver(
    device: &str,
    db_path: Option<&str>,
    recreate: bool,
    timeout_secs: u64,
) -> std::io::Result<ScanResult> {
    // Database setup
    let default_db = format!("{}.sqlite", device.replace(".dll", ""));
    let db_path = db_path.unwrap_or(&default_db);

    if recreate {
        std::fs::remove_file(db_path).ok();
    }
}

```

```

let db = Database::new(db_path)?;
let bon_driver_id = db.get_or_create_bon_driver(device)?;
info!("Database: {}, BonDriver ID: {}", db_path, bon_driver_id);

// Open tuner
let tuner = UnTunedTuner::new(device.to_string(), 200000)?;
info!("Opened BonDriver: {}", device);

// Get available spaces and channels
let num_spaces = tuner.enum_tuning_space().unwrap_or(1);
let mut scanned_channels = Vec::new();

for space in 0..num_spaces {
    let channel_names = tuner.enum_channels(space);
    let num_channels = channel_names.as_ref().map(|c| c.len()).unwrap_or(0);

    info!("Space {}: {} channels", space, num_channels);

    for ch in 0..num_channels {
        // Set channel
        if !tuner.set_channel(space, ch as u32) {
            warn!("Failed to set channel: space={}, ch={}", space, ch);
            continue;
        }

        // Wait for lock and check signal
        tokio::time::sleep(Duration::from_millis(500)).await;
        let signal = tuner.get_signal_level();

        if signal < SIGNAL_THRESHOLD {
            debug!("Low signal: space={}, ch={}, signal={:.1}", space, ch,
signal);
            continue;
        }

        // Analyze TS stream
        let mut analyzer = TsAnalyzer::new();
        let start = Instant::now();
        let timeout = Duration::from_millis(ANALYSIS_DURATION_MS);

        while start.elapsed() < timeout && !analyzer.is_complete() {
            if let Some(packet) = tuner.read_packet().await? {
                analyzer.feed(&packet)?;
            }
        }
    }

    // Extract channel info
    for mut channel_info in analyzer.extract_channels() {
        // BonDriver固有情報を記録
        channel_info.bon_space = Some(space);
        channel_info.bon_channel = Some(ch as u32);

        info!("Found: NID={}, SID={}, TSID={}, name={:?}", 
channel_info.nid, channel_info.sid, channel_info.tsid,

```

```

        channel_info.channel_name);

        scanned_channels.push(channel_info);
    }
}

// Merge results to database
let merge_result = db.merge_scan_results(bon_driver_id, &scanned_channels)?;
db.update_scan_history(bon_driver_id, scanned_channels.len(), true, None)?;

info!("Scan complete: inserted={}, updated={}, disabled={}",
    merge_result.inserted, merge_result.updated, merge_result.disabled);

Ok(ScanResult {
    bon_driver_id,
    total_scanned: scanned_channels.len(),
    merge_result,
})
}

#[derive(Debug)]
pub struct ScanResult {
    pub bon_driver_id: i64,
    pub total_scanned: usize,
    pub merge_result: MergeResult,
}

```

showコマンド ([recisdb-rs/src/commands/show.rs](#)):

```

use prettytable::{Table, Row, Cell};

use serde_json;
use crate::database::Database;

pub fn show_channels(
    db_path: &str,
    dll_path: &str,
    format: &str,
) -> std::io::Result<()> {
    let db = Database::new(db_path)?;
    let bon_driver_id = db.get_bon_driver_id(dll_path)?.unwrap();
    let channels = db.get_channels_by_bon_driver(bon_driver_id)?;

    if format == "json" {
        let json = serde_json::to_string_pretty(&channels)?;
        println!("{}", json);
    } else {
        let mut table = Table::new();
        // ... pretty table formatting
        table.printstd();
    }
}

```

```
    ok(())
}
```

5. TLS認証設計

5.1 証明書構成

```
certs/
├── ca.crt          # CA証明書
├── server.crt      # サーバー証明書
├── server.key       # サーバー秘密鍵
└── client.crt      # クライアント証明書
└── client.key       # クライアント秘密鍵
```

5.2 サーバー設定

```
# recisdb-proxy/recisdb-proxy.toml.example
[tls]
enabled = true
ca_cert = "/etc/recisdb-proxy/certs/ca.crt"
server_cert = "/etc/recisdb-proxy/certs/server.crt"
server_key = "/etc/recisdb-proxy/certs/server.key"
require_client_cert = true
```

5.3 クライアント設定

```
; BonDriver_NetworkProxy.ini
[TLS]
Enabled=1
CaCert=ca.crt
ClientCert=client.crt
ClientKey=client.key

[Server]
Address=192.168.1.100
Port=12345
```

6. 設定ファイル構造

6.1 recisdb-rs 設定

```
# ~/.config/recisdb/config.toml
[database]
path = "~/.local/share/recisdb/channels.sqlite"
auto_backup = true

[scan]
timeout = 30
retry_count = 3
```

6.2 recisdb-proxy サーバー設定

```
# /etc/recisdb-proxy/config.toml
[server]
listen = "0.0.0.0:12345"
max_connections = 10

[tuner]
device = "/dev/pt3video0"
pool_size = 4

[tls]
enabled = true
ca_cert = "/etc/recisdb-proxy/certs/ca.crt"
server_cert = "/etc/recisdb-proxy/certs/server.crt"
server_key = "/etc/recisdb-proxy/certs/server.key"
require_client_cert = true
```

7. 実装フェーズ

Phase 0: プロジェクト構成準備 (1週間)

- 新規crate作成 (recisdb-protocol, recisdb-proxy, bondriver-proxy-client)
- ワークスペース設定更新 (Cargo.toml)
- 依存関係のバージョン調整
- 既存コードのバックアップ

Phase 1: データベース基盤 (2週間) 完了

- recisdb-rs/src/database/ 作成
- SQLiteスキーマ設計 (AutoScan計画を基に改良)
- モデル定義 (rusqlite + recisdb-protocol)
- 基本CRUD操作実装
- merge_scan_results() - スキャン結果マージ
- passive_update_channels() - パッシブスキャン更新
- ユニットテスト

実装ファイル:

- `mod.rs` - Database構造体、エラー型
- `schema.rs` - SQLスキーマ (bon_drivers, channels, scan_history)
- `models.rs` - BonDriverRecord, ChannelRecord, MergeResult
- `bon_driver.rs` - BonDriver CRUD + スキャン設定
- `channel.rs` - Channel CRUD + merge + passive update

成果物:

- データベースモジュール
- ユニットテスト

Phase 2: TS解析モジュール (3週間)

- `ts_analyzer` モジュール実装 (PAT/PMT解析)
- `ts_extractor` モジュール実装 (NID/SID/TSID抽出)
- プラットフォーム差異吸収 (feature-gated)
- シミュレーションモード (テスト用)
- ユニットテスト

成果物:

- TS解析モジュール
- テスト用シミュレータ
- 解析アルゴリズムドキュメント

Phase 3: クライアントCLI拡張 (2週間)

- `scan` コマンド実装
- `show / query` コマンド実装
- `tune` コマンドにDB-backedオプション追加
- データベース-backed `ChannelType` 統合
- 統合テスト

成果物:

- CLIコマンド拡張
- 統合テスト
- 使い方ドキュメント

Phase 4: プロトコル基盤 (1週間) 完了

- `recisdb-protocol` クレート作成
- メッセージ型定義 (AutoScan/Proxy共用)
- バイナリコーデック実装
- ユニットテスト (18件 + doctest 4件 パス)
- `ChannelInfo, ChannelSelector` 型追加
- `broadcast_region.rs` - NID→放送種別・地域判定

成果物:

- プロトコル定義

- コーデック実装
- チャンネル管理型
- 放送地域判定モジュール

Phase 5: プロキシサーバー (3週間) 進行中

- `recisdb-proxy` クレート作成
- TCP/TLSリスナー実装 (基盤)
- `TunerPool` とチャンネル共有ロック
- `tuner/lock.rs` - 排他/共有ロック機構
- `tuner/selector.rs` - フォールバック付きチューナー選択
- `tuner/shared.rs` - 信号レベル・パケット追跡機能追加
- セッション管理 (プロトコル統合)
- データベース統合
- 統合テスト

実装ファイル:

- `tuner/lock.rs` - TunerLock, ExclusiveLockGuard, SharedLockGuard
- `tuner/selector.rs` - TunerSelector, ChannelCandidate, FallbackResult
- `tuner/shared.rs` - lock(), get_signal_level(), has_received_packets()
- `tuner/mod.rs` - 新モジュールのエクスポート

成果物:

- サーバー基盤
- チューナーロック機構
- フォールバック選択
- ユニットテスト (11件パス)
- チャンネル共有機能 (部分)
- 統合テスト (e2e)

Phase 6: クライアントDLL (3週間)

- `bondriver-proxy-client` クレート作成
- IBonDriver vtable実装 (v1/v2/v3)
- リングバッファ実装 (2MB, lock-free)
- TCPクライアント (tokio)
- TLS認証
- DLLビルド設定
- TVTest動作確認

成果物:

- BonDriver互換DLL
- リングバッファ実装
- TVTest動作確認結果

Phase 7: 統合・最適化 (2週間)

- 全機能統合テスト
- パフォーマンス最適化
- ドキュメント更新
- 設定ファイルサンプル作成
- CI/CD設定更新

成果物:

- 完全統合システム
- 性能レポート
- 最終ドキュメント

8. テスト戦略

8.1 ユニットテスト

- **データベース:** モックを使用したCRUD操作テスト
- **TS解析:** パケットシミュレーションによる解析テスト
- **プロトコル:** コーデック単体テスト

8.2 インテグレーションテスト

- **シミュレーション環境:** TSパケット生成によるスキャンテスト
- **TCPプロキシ:** エンドツーエンドテスト
- **TLS認証:** 証明書検証テスト

8.3 手動テスト

- **実HW:** Linux/Windowsでのスキャンテスト
- **TVTest:** プロキシDLLの動作確認
- **負荷テスト:** 複数クライアント同時接続

8.4 テストコマンド例

```
# TS解析テスト（シミュレーション）
cargo test --features ts-analyzer ts_analyzer::

# データベーステスト
cargo test --features database database::

# 統合テスト（シミュレーション環境）
cargo test --features ts-analyzer,database integration::
```

9. 依存関係管理

9.1 ワークスペース Cargo.toml

```
[workspace]
members = [
    "b25-sys",
    "recisdb-rs",
    "recisdb-protocol",
    "recisdb-proxy",
    "bondriver-proxy-client",
]

[workspace.dependencies]
# 共通依存
tokio = { version = "1", features = ["full"] }
tokio-rustls = "0.25"
rustls = "0.22"
serde = { version = "1.0", features = ["derive"] }

# 特定crate用
rusqlite = { version = "0.31", features = ["bundled"] }
bitstream-io = "0.2"
```

9.2 既存依存関係の維持

- `futures-util` → 繙続使用（非同期I/O）
- `nom` → 繙続使用（チャネルパーサー）
- `clap` → 繙続使用（CLIパーサー）

9.3 新規依存関係

- `rusqlite` → データベース
- `serde / serde_json` → シリアライズ
- `prettytable-rs` → テーブル出力
- `bitstream-io` → TSパケット解析
- `tokio-rustls` → TLS通信
- `libloading` → DLL動的ロード（Windows）

10. 設計判断の根拠

決定	理由
ワークスペースレベル依存管理	複数crate間で依存競合を回避
Feature-gated TS解析	実HW依存テストを分離
EnumベースChannelType	既存コードと互換性確保
ロックフリーRingBuffer	高パフォーマンス・低遅延要件
TLS証明書認証	セキュリティ要件（企業ネットワーク対応）
データベース-backed チューニング	パフォーマンス・一貫性確保

11. リスクと緩和策

リスク	影響	緩和策
依存関係競合	ビルド失敗	ワークスペースレベルで依存管理
TS解析が実HW依存	テスト不可能	シミュレーションモード実装
プロキシパフォーマンス	遅延増大	リングバッファ+ロックフリー設計
データベーススケーラビリティ	パフォーマンス劣化	インデックス最適化、WALモード
TLS認証の複雑性	設定ミス	設定確認ツール、詳細ドキュメント
DLLバイナリサイズ	配布サイズ増大	依存関係最適化、stripping

12. 使用例

12.1 チャンネルスキャン

```
# 基本スキャン
recisdb scan --device BonDriver_XXXXXXX.dll

# カスタムDB位置
recisdb scan --device BonDriver_XXXXXXX.dll --database channels.db

# 再構築
recisdb scan --device BonDriver_XXXXXXX.dll --recreate

# ログ表示
RUST_LOG=info recisdb scan --device BonDriver_XXXXXXX.dll
```

12.2 チャンネル表示

```
# テーブル形式
recisdb show --device BonDriver_XXXXXXX.dll

# JSON形式
recisdb show --device BonDriver_XXXXXXX.dll --format json
```

12.3 チャンネルクエリ

```
# クエリ（直接）
recisdb query --device BonDriver_XXXXXXX.dll --channel BS101

# NID/SID/TSID指定
recisdb query --device BonDriver_XXXXXXX.dll --nid 0x0001 --sid 0x0001 --tsid
0x0000
```

```
# マニュアル枝番指定  
recisdb query --device BonDriver_XXXXXXX.dll --manual-sheet 1
```

12.4 プロキシサーバー起動

```
# 基本起動  
cargo run -p recisdb-proxy -- --listen 0.0.0.0:12345  
  
# 設定ファイル使用  
cargo run -p recisdb-proxy -- --config /etc/recisdb-proxy/config.toml  
  
# デバッグモード  
RUST_LOG=debug cargo run -p recisdb-proxy -- --listen 0.0.0.0:12345 --tuner  
/dev/pt3video0
```

12.5 TVTest連携

```
; BonDriver_NetworkProxy.ini  
[TLS]  
Enabled=1  
CaCert=C:\certs\ca.crt  
ClientCert=C:\certs\client.crt  
ClientKey=C:\certs\client.key  
  
[Server]  
Address=192.168.1.100  
Port=12345
```

13. 次のステップ

この計画書を承認いただいたら、Phase 0から実装を開始します。各フェーズの詳細な実装計画は、フェーズ開始時に作成します。

承認が必要な事項

1. 既存コードの変更範囲
2. 新規依存関係の追加
3. ビルド時間増加の許容
4. テスト環境の準備（実HW/HDD）