ANTHROP\C

MCPプロキシを大規模に運用する

Camila Rondinini | Anthropic

この日本語版は、Claude（AI）のみによって翻訳されたものです。誤訳が含まれている可能性が非常に高いです。

This Japanese version was translated solely by Claude (AI) and will very likely contain errors.

自己紹介

Camila Rondinini

カミラ・ロンディニーニ

- From Brazil 🇧🇷 Based in London 🇬🇧
- 日本は初めてです 🇯🇵。各地を巡り、美味しい食事を楽しむのを心待ちにしています
- AnthropicのConnectivityチームでバックエンドエンジニアとして働いています
- 私のチームは、一部のプロダクトがリモートMCPサーバーと通信するために使うMCPプロキシを担当しています。今日はその話をします

私たちのMCPプロキシ

Anthropicの各プロダクトとリモートMCPサーバーの間に置かれたフォワードプロキシです。

本日の内容

このプロキシを大規模に運用する中で向き合った3つのテーマ

01 ユーザーの接続を維持する

02 上流へのリクエストを削減する

03 サーバーの健全性を把握する

第1部

ユーザーの接続を維持する

サーバーを使うための認証は、理想的には一度だけで済むべきです

1 **サーバーが認証を要求**

`401 Unauthorized` 認可サーバーへのポインタ付きで返ります。

2 **認可フロー**

認可URLを組み立て、ユーザーはリダイレクトされ、一度同意し、codeを持って戻ってきます。

3 **トークン交換**

トークンエンドポイントでcodeを交換し、返ってきたものを保存します。

↻ **ユーザーの手を借りないリフレッシュ**

When a `refresh_token`が含まれていれば、期限切れ時にそれで新しいアクセストークンを取得できます。これを繰り返します。

トークンエンドポイント → プロキシ

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "access_token": "eyJhbGciOi...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "8xLOxBtZp8...",
  "scope": "read write offline_access"
}
```

まずリフレッシュトークンを取得する

可能な場合に、まずリフレッシュトークンを取得する

並行実行下でリフレッシュを正しく行う

リフレッシュ失敗が回復可能かどうかを見極める

まずリフレッシュトークンを取得する

サポートの有無は分かる

認可サーバーのメタデータが教えてくれます。`refresh_token`が`grant_types_supported`に含まれていれば、サーバーはリフレッシュフローに対応しています。

```
"grant_types_supported": [
  "authorization_code",
  "refresh_token"
]
```

~4%

のサーバーは、私たちのデータ上では`refresh_token`グラントを公開していないのに発行しています。あくまで目安として扱いましょう。

しかし、サーバーに`refresh_token`を返させる方法は千差万別

どうすれば返してもらえるかは、完全に認可サーバー次第です。

認可URLに特定のパラメータを要求するもの

クライアントメタデータでの宣言を要求するもの
`refresh_token`クライアントの`grant_types`.

自社コンソールでの有効化を要求するもの
クライアントが何を送っても関係ありません。

何も要求しないもの
ただ返ってきます。

具体例 · 1/3

認可URLに特定のパラメータを要求するもの

scopeか追加のクエリパラメータ。しかもプロバイダーごとに異なります。ドキュメントを読むか、リフレッシュトークンが一向に返ってこないことに気づいて初めて分かります。

プロキシ → 認可エンドポイント

```
https://auth.example.com/authorize
  ?response_type=code&client_id=…
  &redirect_uri=…&state=…
  &code_challenge=…&code_challenge_method=S256
```

…さらにプロバイダーによって、次のいずれか：

- `&scope=… offline_access` — Entra、Auth0、Okta、Atlassian（OIDC）
- `&scope=… refresh_token` — Salesforce
- `&prompt=consent` — offline_access付きOIDC、Google
- `&access_type=offline` — Google
- `&token_access_type=offline` — Dropbox
- `&duration=permanent` — Reddit

具体例 · 2/3

一部の認可サーバーは登録済みグラントタイプを厳格に適用し、クライアントメタデータの refresh_token に grant_types

Dynamic Client Registration（DCR）とClient ID Metadata Document（CIMD）は、RFC 7591で定義された同じクライアントメタデータフィールドを共有します。`grant_types` はその一つで、省略時のデフォルトは `authorization_code` です。

厳格なサーバーはこれを文字通り解釈し、宣言していないクライアントにはリフレッシュトークンを発行しません。

プロキシ → 認可エンドポイント

```
https://auth.example.com/authorize
  ?client_id=https://claude.ai/oauth/mcp-oauth-client-metadata
  &…
```

↓ client_idがCIMDのURLです。認可サーバーがこれを取得します：

GET https://claude.ai/oauth/mcp-oauth-client-metadata

```
{
  "client_id": "https://claude.ai/oauth/mcp-oauth-client-metadata",
  "client_name": "Claude",
  "client_uri": "https://claude.ai",
  "redirect_uris": ["https://claude.ai/api/mcp/auth_callback"],
  "grant_types": ["authorization_code", "refresh_token"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "none"
}
```

具体例 · 3/3

そして、クライアントが何を送っても変わらないこともある

プロバイダー側で誰かが設定を切り替えるか、あるいは何もしなくてもサーバーが返してくれるか、のどちらかです。

リクエストの外側

プロバイダーのコンソールにあるスイッチ

誰かがサーバー側でアプリのリフレッシュトークン（またはトークンローテーション、有効期限）を有効化する必要があります。それまでは、どんなパラメータを追加してもレスポンスは変わりません。

Slackのトークンローテーション · GitHub Appsのトークン有効期限 · Auth0の「Allow Offline Access」

常に返される

サーバーはアクセストークンと一緒に常にリフレッシュトークンを返します。scopeもパラメータも設定も不要です。

HubSpot · Asana · Zoom · Box

まとめ

LLMクライアントは初めての「ユニバーサル」なOAuthクライアントであり、あらゆるサーバーの癖に対応することが期待されている

これらの癖はどれも新しいものではありません。違うのは、MCPクライアントはどんなMCPサーバーともそのまま動くべきだという期待です。そのためには、各プロバイダーの癖をできる限りすべて扱えるようにしておく必要があります。

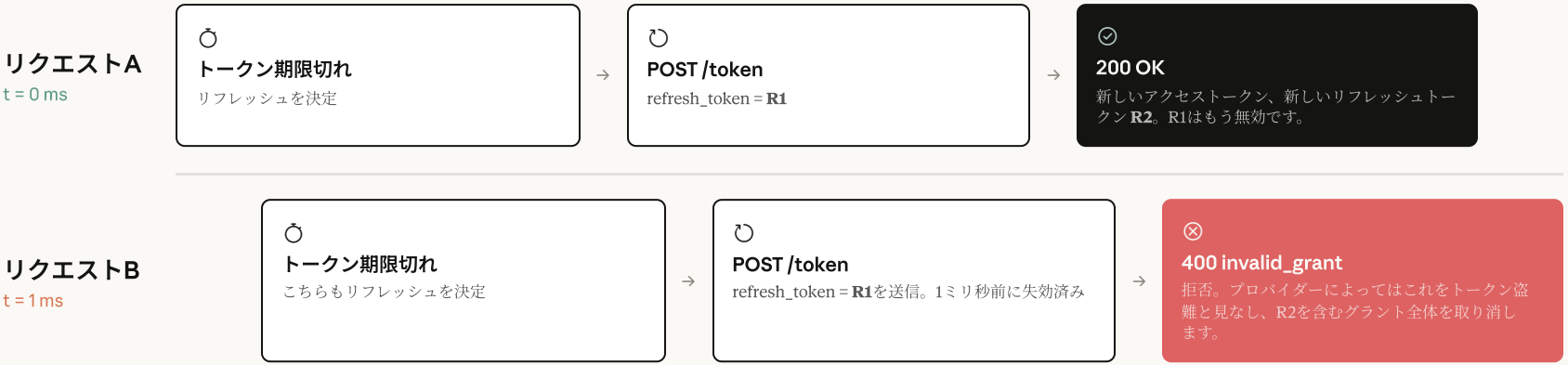
並行実行下でリフレッシュを正しく行う

可能な場合に、まずリフレッシュトークンを取得する

並行実行下でリフレッシュを正しく行う

リフレッシュ失敗が回復可能かどうかを見極める

並行するリフレッシュは、守ろうとしているトークンを殺してしまう



問題なく接続していたユーザーが「再接続してください」と言われる。原因はすべてプロキシ側です。

利用量に比例します。ヘビーユーザーほど確実に失敗します。

並行実行下でリフレッシュを正しく行う

分散ロックでリフレッシュを一度に一つに絞る

クレデンシャル単位の短寿命ロック。ロックを持つ者だけが認可サーバーと話し、他はストアを読みます。

↔ 10件の並行リクエストに対し、認可サーバーへの呼び出しは1回。

Aのリフレッシュが失敗しても同じ。Aが失敗を書き込み、Bはそれを返す。やはり1回です。

並行実行下でリフレッシュを正しく行う

ロックは並行リクエストによる無効化を防ぐ

早めにリフレッシュ

トークン寿命の最後の約5%で（妥当な下限・上限付き）。一つのリクエストがリフレッシュする間、他はまだ有効なトークンを持っています。

待機側に2種類のタイムアウト

有効なトークンをまだ持っている → 数秒待ってから、そのトークンで続行。
トークンが既に無効 → リースの満了まで待つ。それしか選択肢がありません。

待機側は成功だけでなく失敗も読む

新トークンだけを探していると、リフレッシュ失敗時に全員がまたリフレッシュに走ります。ロック保持者がエラーを記録し、待機側はそれを見てバックオフします。

リフレッシュ失敗が回復可能かどうかを見極める

可能な場合に、まずリフレッシュトークンを取得する

並行実行下でリフレッシュを正しく行う

リフレッシュ失敗が回復可能かどうかを見極める

理論上は

OAuthのRFCは、リフレッシュ失敗の形と各エラーの意味を正確に定義している

RFC 6749 §5.2：トークンエンドポイントは `400`（不正なクライアントには `401`）を返し、JSONボディの `error` は6つの値のいずれか。RFC 6750はリソースリクエストに対して同じことを定めています。

すべてのサーバーがこれに従えば、「リトライか再認証か」は表を引くだけで済みます。

RFC 6749 §5.2 · トークンエンドポイントのエラー

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{ "error": "invalid_grant", "error_description": "…" }
```

6つの値と、それがクライアントに伝えること

`invalid_grant`	リフレッシュトークンが無効 → ユーザーに再認証を求める
`invalid_client`	クライアント登録に問題がある
`unauthorized_client`	このクライアントはこのグラントを使えない
`invalid_request`	リクエストの組み立てが間違っている
`unsupported_grant_type`	サーバーはリフレッシュ自体に非対応
`invalid_scope`	付与された以上を要求した

実際には

リフレッシュ失敗の約25%は、OAuth準拠のエラーのどれにも当てはまらない

74%

仕様準拠

72.5% 正規の§5.2コード付き400

1.4% 401 + `invalid_client`、仕様が許す唯一の401

10%

HTTPサーバーエラー

9.3% OAuth形式でないボディやHTML付きの5xx

0.9% Cloudflareの52x / 530エッジエラー

16%

その他すべて

6.5% 404、エンドポイントの消失や移動

5.4% 誤ったコードや独自コード付きの401 / 403

3.3% ボディがOAuthエラーでない、またはベンダー独自コードの400 / 429

0.6% トークンなしの2xx、迷い込んだ405 / 410 / 415…

リフレッシュ失敗が回復可能かどうかを見極める

理想的には、リフレッシュのエラーが回復可能かどうかを教えてくれる

サーバーが仕様に準拠していれば、おおむね教えてくれます：

`invalid_grant`	リフレッシュトークンはもう無効。二度とリフレッシュできません。	⊗ 無効。再認証。
`invalid_client`	クライアントの登録方法によります：	
DCR	登録が消え、トークンも道連れです。	⊗ 無効。再認証。
CIMD	おそらくメタデータ文書の取得が一時的に失敗しただけ。	↻ リトライ。
事前登録	シークレットのローテーションかアプリの無効化。クレデンシャルの所有者なら直せます。	回復可能。ただしユーザーではなく。
5xx	単純なHTTPサーバーエラー。ときに `Retry-After` ヘッダー付き。	↻ バックオフ付きでリトライ。
非準拠のものすべて	リフレッシュトークン取得と同じ「癖」の世界に戻ります。一つずつ学び、解釈するしかありません。	? 場合による。

認可サーバー開発者の皆さんへ

OAuth準拠のエラーを使い、その意味を文字通りに守ってください。それがクライアントが回復可能かどうかを判断できる唯一の手がかりです。

リフレッシュ失敗が回復可能かどうかを見極める

理想的には、エラーが回復可能かどうかを教えてくれる

サーバーが仕様に準拠していれば、おおむね教えてくれます：

`invalid_grant`	リフレッシュトークンはもう無効。二度とリフレッシュできません。	⊗ 無効。再認証。
`invalid_client`	クライアントの登録方法によります：	
DCR	登録が消え、トークンも道連れです。	⊗ 無効。再認証。
CIMD	おそらくメタデータ文書の取得が一時的に失敗しただけ。	↻ リトライ。
事前登録	シークレットのローテーションかアプリの無効化。クレデンシャルの所有者なら直せます。	回復可能。ただしユーザーではなく。
5xx	単純なHTTPサーバーエラー。ときに `Retry-After` ヘッダー付き。	↻ バックオフ付きでリトライ。
非準拠のものすべて	リフレッシュトークン取得と同じ「癖」の世界に戻ります。一つずつ学び、解釈するしかありません。	? 場合による。

認可サーバー開発者の皆さんへ：OAuth準拠のエラーを使い、その意味を文字通りに守ってください。それがクライアントが回復可能かどうかを判断できる手がかりです。

リフレッシュ失敗が回復可能かどうかを見極める

この場合、リトライは効くのか？

リフレッシュ時に実際に受け取ったエラーの例

```
# POST /oauth/token  grant_type=refresh_token

HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 67

{
  "error": "server_error",
  "error_description": "..."
}
```

ステータス：準拠
RFC 6749 §5.2は400と定める

形式：準拠
JSONボディに `error` フィールドあり

値：/tokenのリストにはなく、/authorize用のみ
§5.2がトークンエンドポイントで許すのはこれだけ：

`invalid_request`	不正な形式のリクエスト
`invalid_client`	クライアント認証の失敗
`invalid_grant`	トークンの期限切れ、取消、または使用済み
`unauthorized_client`	このクライアントはこのグラントを使えない
`unsupported_grant_type`	サーバーはリフレッシュ非対応
`invalid_scope`	許可されていないscope

リフレッシュ失敗が回復可能かどうかを見極める

この場合は、はい。リトライは効きました

しかし非準拠のエラー値では、トークンが本当に失効したのか、サーバーが一時的に不調だっただけなのかを見分けるのが困難です。

```
# POST /oauth/token  grant_type=refresh_token

HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 67

{
  "error": "server_error",
  "error_description": "..."
}

↻ 2  にリトライ →  HTTP/1.1 200 OK
```

ステータス：準拠
RFC 6749 §5.2は400と定める

形式：準拠
JSONボディに`error`フィールドあり

値：/tokenのリストにはなく、/authorize用のみ
§5.2の6つの値のどれでもないため、レスポンスだけでは判断できません。

リフレッシュ失敗が回復可能かどうかを見極める

エラーの種類ごとに扱いを分け、最終判断はサーバーの 401 に委ねる

準拠エラー、既知の非準拠の癖、その他すべてに、それぞれ別の失敗処理を用意します。リフレッシュが何と言おうと、MCPサーバーが拒否するまでトークンは有効です。

準拠エラー

自信を持って読める

`invalid_grant` は確定。`invalid_client` はクライアントの登録方法次第。5xxは一時的。レスポンスがどれかを教えてくれます。

非準拠サーバー固有のエラー

学習し、確定エラーとして扱う

一部の認可サーバーは、取消・期限切れのグラントに対し独自のエラーを返します。一度そのサーバーのエラーを把握すれば、それを認識して即座にリトライを止められます。

その他すべて

バックオフし、しばらく試し、それから諦める

一定期間、バックオフ付きでリトライ。それでもリフレッシュが失敗し、サーバーが依然401を返すなら、接続を無効としてユーザーをサインアウトします。

助けになること

準拠エラーが増えれば、無駄なリフレッシュが減る

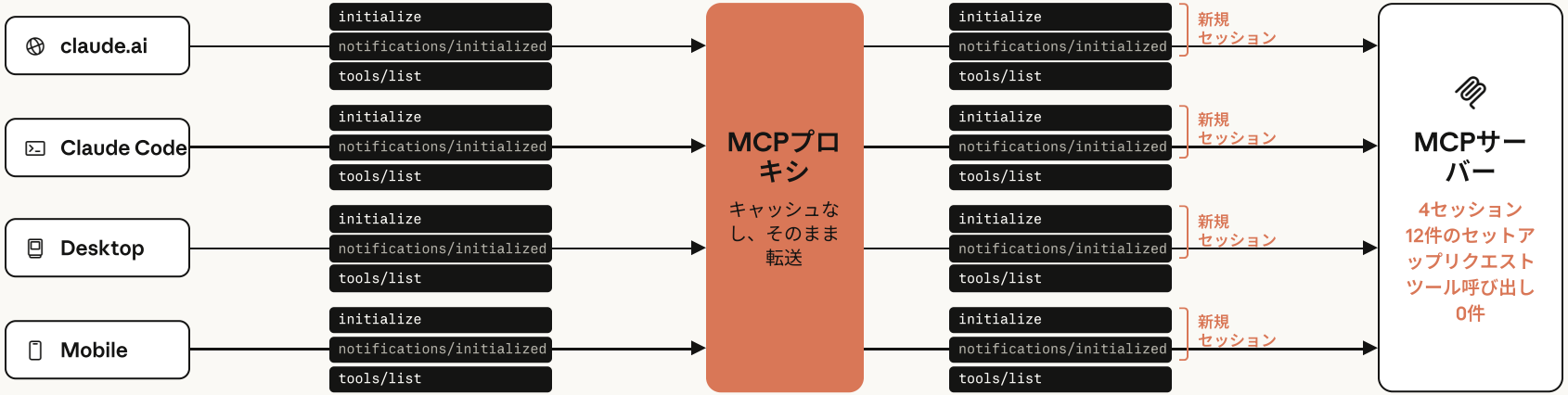
より多くの認可サーバーが仕様のエラーコードを文字通りに使えば、クライアントは即座に判断でき、成功するはずのないリフレッシュリクエストを送らずに済みます。

第2部

上流へのリクエストを削減する

上流に送っていたものの大半は無駄なセットアップリクエストだった

すべてのサーフェスが `initialize` → `notifications/initialized` → `tools/list` をユーザー×サーバーごとに繰り返す。さらに同じ `tools/list` が絶えず再送され、私たちはそのすべてを転送していました。



1ユーザー、1サーバー。どれもユーザーの役には立たず、しかも自社のダッシュボードより先にパートナーから指摘されました。

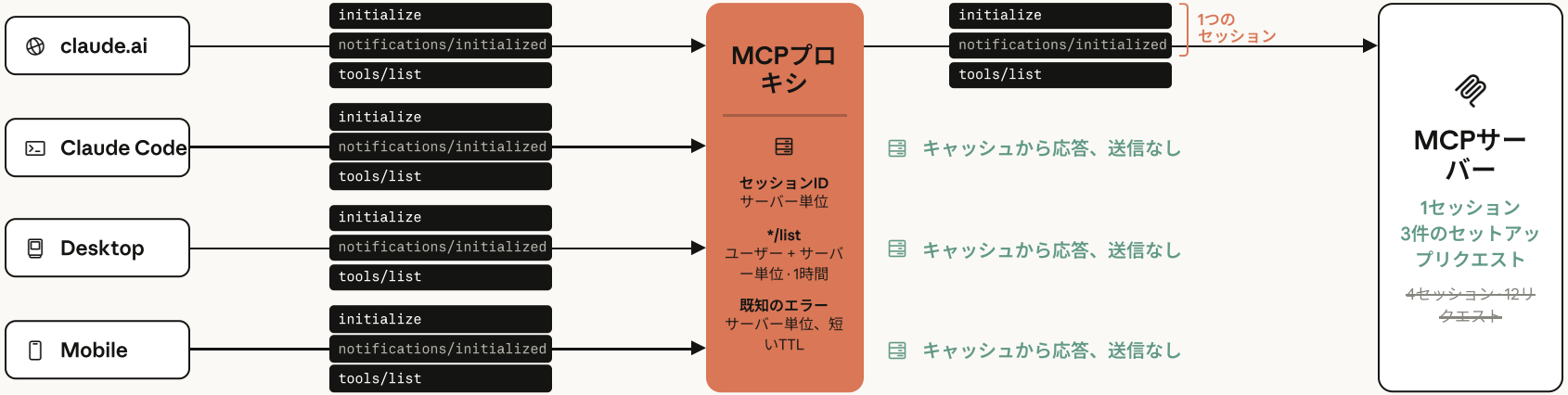
そこでプロキシでキャッシュを始めた

リストがどれだけ有効かサーバーは教えてくれないため、1時間を中間点に選びました。キャッシュする価値があり、同日中の変更も拾えます。

MCPセッションID ユーザー + サーバー単位

すべての `*/list` の結果 ユーザー + サーバー単位 · TTL 1時間

既知の故障サーバー サーバー単位 · 短いTTL



クライアントには何も変わらず、サーバーにはすべてが変わりました。

しかし、これでもまだ不十分

リストをユーザー単位でキャッシュしているのは、仕様上ツールがユーザーごとに異なる *可能性がある* からです。実際にはほとんど異なりません。

ログにあるすべての `tools/list` レスポンスの決定的ハッシュを取り、同じサーバーのユーザー間で比較しました。

つまりユーザー単位のキーは、起きもしないケースのためにキャッシュエントリと上流呼び出しを増やしているだけです。理想はサーバー単位でキャッシュし、本当にユーザー固有のツールがある場合だけユーザー単位にすることです。

ログから

~92%

のサーバーが全ユーザーに同じツールリストを返す

全ユーザーで `tools/list` ハッシュが一致。

2026-07-28の仕様改訂で、サーバー単位のキャッシュ粒度が可能になりました！

サーバーが、リストが全員に共通か（public）、どれだけキャッシュしてよいかを教えてくれます。

2026-07-28の仕様改訂以降

サーバーのレスポンスがキャッシュの方法を教えてくれる

ttlMs
レスポンスを再利用してよい期間。

cacheScope: "public"
共有プロキシはどのユーザーにも返してよい。サーバー単位のキャッシュ。

cacheScope: "private"
クレデンシャル間で共有不可。ユーザー単位のキャッシュ。トークンのバイト列ではなく保存済み接続をキーにするため、リフレッシュ後も残ります。

tools/list · JSON-RPCのresultボディ内、ヘッダーではない

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "complete",
    [ … ]
    "ttlMs": 300000,
    "cacheScope": "public"
  }
}
```

同じ改訂でセッションと `initialize` ハンドシェイクが廃止されました。セッションキャッシュはもう過去のもの。プロトコルにもう存在しないコストへの回避策でした。

第3部

サーバーの健全性を把握する

MCPサーバーはさまざまな層で失敗する

ネットワーク	**一切応答しない** DNS解決失敗、接続拒否、TLS失敗、タイムアウト。HTTPレスポンスなし。	`ENOTFOUND` `ECONNREFUSED` `ETIMEDOUT`
HTTP	**エラーステータスで応答** サーバーまたはその前段からの本物の4xx / 5xx。	`HTTP 502 Bad Gateway`
JSON-RPC	**200でJSON-RPCエラーを返す** プロトコルがリクエスト失敗を告げる：不明なメソッド、不正なパラメータ、サーバーの例外。 resultなし。モデルには届きません。	`200 {"error": {"code": -32603, "message": …}}`
ツール結果	**200でisErrorを立てて返す** 呼び出しは成功。ツールが、実行しようとした処理の失敗を報告しています。 これはモデルに向けたものです。	`200 {"result": {"content": […], "isError": true}}`

一部のエラーコードは文脈で意味が変わる

同じステータスでも、発生時点と送った内容によって、日常的なものにも本当の問題にもなります。

404

セッション終了

サーバーが知らないセッションIDを破棄した。新しいセッションを開始して続行。

または

エンドポイントが存在しない

URLの誤り、またはサーバーの移転。設定を変えるまで何も動きません。

401

トークンが無効

トークンを持っていて送ったが、サーバーが拒否した。リフレッシュするか、ユーザーに再認証を求める。

または

トークンを送っていない

内部で取得できず、トークンなしで上流に送った。これは私たちの問題です。

文脈を理解することが、MCPサーバーの健全性を正しく読み取る鍵です。

これらを正規化するため、すべての失敗を `error_code` と `fault`

現在使っているマッピングの一部です。全リストはもっと長く、ケースを追加するにつれ増え続けています。

ステータス	文脈	error_code	fault
401	保存済みグラントが無効。リフレッシュでは救えない	`mcp_invalid_oauth_token`	`user_auth_expired`
	ユーザーはこのサーバーに接続したことがない · エラー率から除外	`mcp_unauthorized_no_token`	`expected_noauth`
404	URLがMCPサーバーではなくWebサイトのように応答する	`mcp_endpoint_not_found`	`misconfiguration`
400	ハンドシェイク前にストリームが開かれた	`mcp_session_id_missing`	`misconfiguration`
502	サーバーが接続を切断、またはストリーム途中で不正な応答	`mcp_connection_error`	`upstream_server`
500	保存済みトークンを内部で取得できなかった · リクエストは上流に送られていない	`mcp_token_fetch_failed`	`internal`
429	認可サーバーがリフレッシュをレート制限した · バックオフ。トークンはまだ使えるかもしれない	`mcp_upstream_auth_rate_limited`	`upstream_auth_server`

リクエストごとに1行の構造化ログを使う

メトリクスに収まらないすべて：メソッド、クレデンシャルの状態、サーバーの応答、私たちの応答。

匿名化し、サーバーとソースをキーにします。ディレクトリのサーバーは審査済みなので、そのエラーは本物の障害。ユーザー追加のものには設定ミスのノイズが混じります。

まだ足りないもの

これをサーバー開発者と共有すること。MCPプロキシをゼロから作るなら、初日から組み込みます。

```
{
  // プロキシしたリクエストごとに1イベント
  "event": "mcp_proxy_request_completed",
  "mcp_method": "tools/call",
  "server_host": "mcp.example.com",          // ハッシュ  または          なURLは    しない
  "installation_source": "directory",        // directory | user_added
  "client": "web_app",                       // どのプロダクトからの  び  しか

  //
  "outcome": "error",                        // success | error
  "error_code": "mcp_invalid_oauth_token", //    きで       び  し  にもヘッダーで  す
  "fault_domain": "user_auth_expired",       // none | internal | upstream_server | upstream_auth_server
                                             // | user_auth_expired | misconfiguration | expected_noauth
  "exclude_from_error_rate": false,          //        ユーザー」の401のみtrue

  // どのクレデンシャルで  ったか
  "credential": {
    "outcome": "has_token",                  // has_token | no_token | invalid_token | store_unavailable
    "refresh_outcome": "failed",             // skipped_not_needed | succeeded | failed | skipped_backoff …
    "has_refresh_token": true,
    "expires_in_seconds": -31                //        で  に     れ
  },

  // MCPサーバーの
  "upstream": {
    "status_code": 401,
    "content_type": "application/json",
    "retry_after": null,
    "jsonrpc_error_code": null,              // 200のボディがJSON-RPCエラーのときに
    "tool_result_is_error": null             // tools/callがisError: trueを  したときに
  },

  //  び  し  に  した
  "response": { "status_code": 401 }, //   じステータス。    はcredentialブロックに

  // キャッシュ
  "cache_result": "skipped"                  // hit | stale | miss | skipped  tools/callはキャッシュしない
}
```

最後に

まとめ

まとめ

MCP認可サーバー開発者に伝えたいこと

仕様が推奨するとおり、DCRではなくCIMDか事前登録クライアントを使う

DCRでは、自分が作ったわけでも連絡できるわけでもないクライアントのデータベースを抱えることになります。

- 登録は期限切れや削除で消え、クライアントに見えるのは `invalid_client`.
- CIMDでは `client_id` は取得してキャッシュするURLです。保存するものはなく、クライアントが誰かも分かります。
- 既知のクライアントが少数なら、静的なIDとシークレットの方がどちらよりも簡単です。

エラーはOAuthに準拠させる

プロキシはエラーを認識できる速さでしか反応できません。

- 仕様のステータスコードと `{"error": …}` のJSON形式を使う。
- コードの意味を文字通りに守る：`invalid_grant` はグラントが本当に無効なときだけ。
- ダウンしているならそう伝える：`503` または `429` + `Retry-After`.

まとめ

今日MCPプロキシを作るなら、こうする

リフレッシュを正しく設計する
リフレッシュトークンの取得方法に注意する。自分のリクエスト同士が競合しないよう、分散ロックの背後でリフレッシュする。リフレッシュに必要なOAuthの状態はすべて最初から保存する。

キャッシュで無駄な呼び出しを避ける
新仕様のキャッシュ機能を活用する。

構造化ログ
リクエストごとに1行、すべてを詰め込む。何が起きているかが見えるように。

独自の、文脈を持つエラーコード
それぞれに責任領域を付ける：クライアント、プロキシ、上流サーバー、認可サーバー。

サーバーの健全性を、初日からサーバー開発者に提供できるものにする
MCPサーバーが失敗しうるすべての層を考え抜き、その情報がサーバー運営者に戻せるように作る。

今日触れられなかった2つ

サーバー単位の認可サーバーメタデータ上書きを組み込む
独自エンドポイントや誤ったメタデータを持つサーバーは多い。サーバー単位のオプションを最初から用意する。

認証ディスカバリーをキャッシュする
well-known文書とメタデータはユーザー間で変わらない。サーバーをキーにし、TTLを付ける。

ANTHROP\C

ありがとうございました

ありがとうございました

Camila Rondinini

MCP Connectivity · Anthropic、ロンドン