

A2A プロトコルを試してみる

さくらのAI Meetup vol.11 「Agent2Agent (A2A)」

2025/06/25(水)

自己紹介

- azukiazusa
- <https://azukiazusa.dev>
- FE (フロントエンド|ファイアーエムブレム)
が好き



Agent2Agent (A2A) プロトコルとは

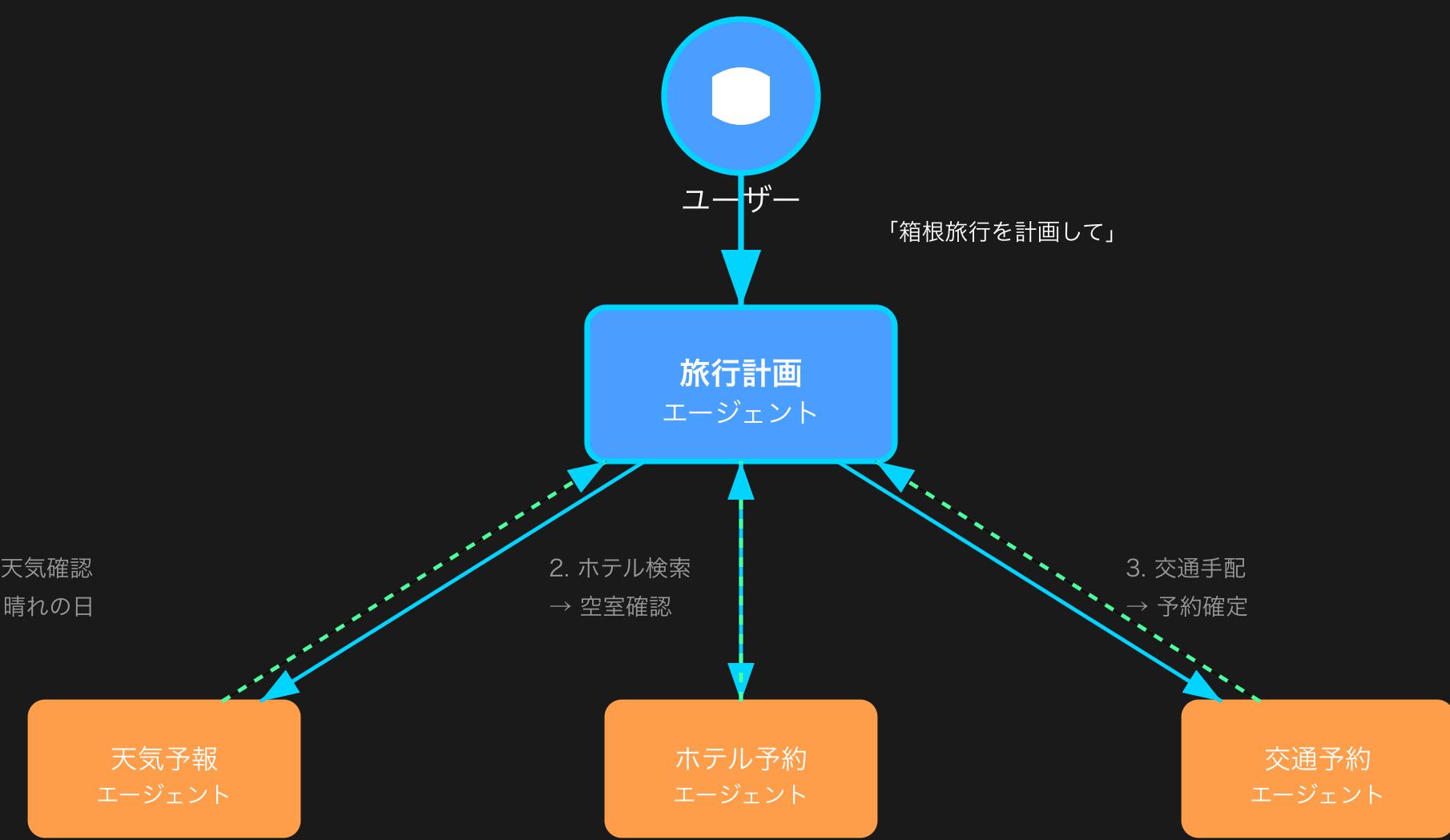
AI エージェント間の連携を標準化するプロトコル

- Google が開発・発表
- 異なるベンダーのエージェント同士が連携
- 標準的な HTTP 上に構築

なぜ A2A が必要か？

AI エージェントが効果的に目的を達成するためには、多様なエージェントがエコシステム内で連携できることが重要

- 旅行計画の例：
 - 天気予報を調べる
 - 宿泊先を予約する
 - 交通機関を予約する



統合された旅行プラン

A2A の 3つのアクター



AgentCard

エージェントの機能を記述するメタデータ

- エージェントの名前・説明
- サポートされている機能（ストリーミング、プッシュ通知）
- 提供するスキル
- 認証メカニズム
- `/.well-known/agent.json` でホスト

AgentCard の実装

```
export const agentCard: AgentCard = {  
  name: "Dice Agent",  
  description: "サイコロを振るエージェント",  
  url: "https://localhost:3000",  
  version: "1.0.0",  
  defaultInputModes: ["text/plain"],  
  defaultOutputModes: ["text/plain"],  
  capabilities: {  
    streaming: true,  
    pushNotifications: false,  
  },  
  skills: [{  
    id: "dice-roll",  
    name: "diceRoll",  
    description: "ランダムな数字を生成するサイコロツール",  
    example: [  
      "サイコロを振ってください",  
      "12面のサイコロを振ってください",  
    ],  
    tags: ["dice", "random"],  
    inputModes: ["text/plain"],  
    outputModes: ["text/plain"],  
  }],  
};
```

AgentCard の実装

```
import { Hono } from 'hono'
import { agentCard } from './agentCard';
const app = new Hono()

app.get("/.well-known/agent.json", (c) => {
    return c.json(agentCard);
});
```

Task オブジェクト

クライアントとサーバー間の通信を管理

- クライアントは `Message` にプロンプトを含めて送信
- サーバーはタスクの現在の状態（作業開始・完了・失敗など）を更新して返す
- 最終結果として `Message` や `Artifact` を返す

Task の状態遷移

- `submitted` : サーバーにリクエストが受理された
- `working` : エージェントによりタスクの処理が開始された
- `input-required` : 追加の入力が必要
- `completed` : 完了
- `canceled` : キャンセル
- `failed` : 失敗
- `rejected` : 拒否
- `auth-required` : 認証必要
- `unknown` : 不明な状態

JSON-RPC 形式の通信

リクエストとレスポンスは JSON-RPC 2.0 形式で行われる。POST リクエストで送信

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "method": "message/send",  
    "params": {  
        "id": "uuid",  
        "message": {  
            "role": "user",  
            "parts": [{"type": "text", "text": "サイコロを振って"}]  
        }  
    }  
}
```

レスポンス例

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": {  
        "id": "de38c76d-d54c-436c-8b9f-4c2703648d64",  
        "sessionId": "a3e5f7b6-3e4b-4f5a-8b9f-4c2703648d64",  
        "status": {  
            "state": "completed",  
            "message": [  
                {  
                    "role": "agent",  
                    "parts": [{"type": "text", "text": "サイコロの目は 1 でした"}]  
                }  
            ]  
        },  
    }  
}
```

プロトコルRPCメソッド

定義済みの `method` に対応する `params` を含むリクエストを送信

- `message/send` : メッセージを送信
- `message/stream` : メッセージをストリーミング送信
- `tasks/get` : タスクを送信
- `tasks/cancel` : タスクをキャンセル

サーバーを実装してみる

- エンドポイント `/` で POST リクエストを受け付ける
- JSON-RPC 2.0 形式のリクエストか検証

```
import { Hono } from "hono";
const taskApp = new Hono();

taskApp.post("/", async (c) => {
  const body = await c.req.json();
  if (!isValidJsonRpcRequest(body)) {
    const errorResponse = {
      code: -32600,
      message: "Invalid Request",
    };
    return c.json(errorResponse, 400);
  }
  // リクエストの処理ロジックをここに実装
});
```

body の method を確認

```
taskApp.post("/", async (c) => {
  const body = await c.req.json();
  // ...
  switch (body.method) {
    case "message/send":
      return handleSendMessage(c, body);
    case "tasks/get":
      return handleGetTask(c, body);
    // 省略
    default:
      const errorResponse: ErrorMessage = {
        code: -32601,
        message: "Method not found",
      };
      return c.json(errorResponse, 404);
  }
});
```

params の検証

```
async function handleSendMessage(c: Context, body: any) {  
  const params: MessageSendParams = body.params;  
  // params の検証  
  if (!params || !params.id || !params.message) {  
    const errorMessage: ErrorMessage = {  
      code: -32602,  
      message: "Invalid params",  
    };  
    return c.json(errorMessage, 400);  
  }  
  
  const getOrCreateTaskResult = getOrCreateTask(params.id, params.message);  
}
```

エージェントを呼び出し結果を返す

```
async function handleSendTask(c: Context, body: any) {
  const getOrCreateTaskResult = getOrCreateTask(params.id, params.message);
  // タスクの状態を "working" に更新する
  taskStore.set(params.id, {});
  // AI エージェントを呼び出す
  const result = await generateText({...});
  // タスクの状態を "completed" に更新する
  taskStore.set(params.id, {
    status: {
      state: "completed",
      timestamp: new Date().toISOString(),
      message: [
        {
          role: "agent",
          parts: [{ type: "text", text: `サイコロの目は ${result} でした` }],
        },
      ],
    },
  });
  return c.json({...})
}
```

クライアント実装

```
export class A2AClient {
    async sendMessage(params: MessageSendParams): Promise<any> {
        const response = await fetch(`.${this.baseUrl}/`, {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({
                jsonrpc: "2.0",
                method: "tasks/send",
                params,
                id: crypto.randomUUID(),
            })
        });
        return response.json();
    }

    agentCard(): Promise<AgentCard> {
        return fetch(`.${this.baseUrl}/.well-known/agent.json`)
            .then((res) => res.json());
    }
}
```

CLI ツールの作成

```
const agentCard = await client.agentCard();
const tools: ToolSet = {};

for (const skill of agentCard.skills) {
  tools[skill.id] = tool({
    description: skill.description,
    execute: async ({ input }) => {
      return await client.sendTask({
        message: { role: "user", parts: [{ type: "text", text: input }] }
      });
    }
});
```

main() 関数の実装

```
async function main() {
  const input = await lr.question("You: ");
  const response = await generateText({
    messages: [{ role: "user", content: input }],
    tools, // エージェントカードを元に定義したツールをセット
  });

  console.log(`AI: ${response.message.parts[0].text}`);
}
```

実行結果

あなた： サイコロを振って

AI： サイコロを振った結果は 4 でした！

もう一度振りますか？

JavaScript SDK

Google が提供する公式 SDK で A2A エージェント開発を簡略化

<https://github.com/google-a2a/a2a-js>

```
npm install @a2a-js/sdk
```

AgentExecutor クラスを継承する

AgentExecutor を継承してカスタムエージェントを作成

```
import { AgentExecutor } from "@a2a-js/sdk";

export class DiceAgent extends AgentExecutor {
  public cancelTask = async (
    taskId: string,
    eventBus: ExecutionEventBus,
  ): Promise<void> => {
    // 実行中のタスクをキャンセルするロジックを実装
  };
  async execute(
    requestContext: RequestContext,
    eventBus: ExecutionEventBus
  ): Promise<void> {
    // message/send or message/stream が呼び出されたときの処理を実装
  }
}
```

execute メソッドの実装

```
// クライアントのリクエストを受け取る
const { taskId, message } = requestContext;

// 新しいタスクを作成してクライアントに通知する
eventBus.publish({ id: taskId, status: { state: "submitted" } });
// タスクの状態を "working" に更新
eventBus.publish({ id: taskId, status: { state: "working" } });

const result = generateText(message.parts[0].text);
// タスクの状態を "completed" に更新して結果を返す
eventBus.publish({
  id: taskId,
  status: {
    state: "completed",
    message: [{ role: "agent", parts: [{ type: "text", text: `サイコロの目は ${result} でした` }] }],
  },
});
```

サーバーを起動

```
import { InMemoryTaskStore, DefaultRequestHandler, A2AExpressApp } from "@a2a-js/sdk";
import express from "express";

const taskStore: TaskStore = new InMemoryTaskStore();
const agentExecutor: AgentExecutor = new DiceAgent();

const requestHandler = new DefaultRequestHandler(
  agentCard, // AgentCard の定義
  taskStore,
  agentExecutor
);

const appBuilder = new A2AExpressApp(requestHandler);
const expressApp = appBuilder.setupRoutes(express(), '');

expressApp.listen(3000, () => {
  console.log("A2A server is running on https://localhost:3000");
});
```

クライアントの実装

A2AClient を使ってエージェントと通信

```
import { A2AClient } from "@a2a-js/sdk";
import { randomUUID } from "node:crypto";

const serverUrl = "http://localhost:41241";
const client = new A2AClient(serverUrl);

const result = await client.sendMessage({
  id: randomUUID(),
  message: {
    role: "user",
    parts: [{ type: "text", text: "サイコロを振って" }],
  },
});
```

Mastra での A2A サポート

TypeScript AI エージェントフレームワーク

- **自動対応:** 特別な設定不要
- **クライアント SDK:** `@mastra/client-js`
- **ストリーミング:** SSE による実時間更新

Mastra サーバーのセットアップ

```
npx create-mastra@latest my-mastra-app  
cd my-mastra-app  
npm run dev
```

- `./.well-known/{agentId}/agent.json` でAgentCard公開

Mastra エージェントの作成

```
export const travelAgent = new Agent({  
  name: "travel-agent",  
  instructions: "旅行プランを提案するエージェント",  
  model: anthropic("claude-4-sonnet-20250514"),  
  tools: { weatherTool }  
});
```

Mastra のエントリーポイント

```
import { Mastra } from "@mastra/core";
import { travelAgent } from "./agents/travelAgent";

export const mastra = new Mastra({
  agents: { travelAgent },
});
```

Mastra クライアント

```
npm install @mastra/client-js
```

エージェントカードを取得

```
const client = new MastraClient({  
  baseUrl: "http://localhost:4111"  
});  
  
const a2a = client.getA2A("travelAgent");  
const agentCard = await a2a.getCard();
```

Mastra でのメッセージ送信

```
const response = await a2a.sendMessage({  
  id: crypto.randomUUID(),  
  message: {  
    role: "user",  
    parts: [{ type: "text", text: "箱根旅行プランを提案して" }]  
  }  
});
```

ストリーミング通信

```
const response = await a2a.sendAndSubscribe({
  id: crypto.randomUUID(),
  message: { /* メッセージ */ }
});

const reader = response.body?.getReader();
while (true) {
  const { done, value } = await reader.read();
  if (done) break;
  console.log(new TextDecoder().decode(value));
}
```

まとめ

- A2A はエージェント間連携の標準プロトコル
- エージェントカードでエージェントの機能やスキルを公開
- タスクオブジェクトでクライアントとサーバー間の通信
- JSON-RPC 形式でリクエストとレスポンスをやり取り
- JavaScript SDK や Mastra で A2A に準拠したエージェントを簡単に実装可能

參考資料

- <https://google.github.io/A2A/>
- <https://github.com/google-a2a/a2a-samples/tree/main/samples>
- <https://github.com/google-a2a/a2a-js>
- <https://mastra.ai/>

