MCP Mimarisi

MCP'nin mesaj biçimi olan JSON-RPC 2.0, stdio ve Streamable HTTP transport'ları, initialize/discover/invoke bağlantı yaşam döngüsü, ve capability negotiation.

Orta 20 dk

Son güncelleme: 5 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/mcp-architecture

---

MCP Mimarisi

"MCP'ye Giriş" üç rolü -- host, client ve server -- ve bir server'ın sunabileceği üç primitive'i anlatmıştı. Bu ders bir katman daha derine iniyor: bir client ile server arasında gerçekte ne, hangi sırayla, ve ne tür bir bağlantı üzerinden gidip geliyor. Hiçbiri daha önce kurulan hiçbir şeyi değiştirmiyor -- bu, onun altındaki mekanik katman, ve bu kategorinin son dersi olan "TypeScript ile MCP Sunucusu Oluşturma"nın gerçek, çalışan kodla doğrudan alıştıracağı şey.

Kavramlardan Wire Format'a: JSON-RPC 2.0

Bir client ile server'ın değiş tokuş ettiği her mesaj bir **JSON-RPC 2.0** mesajıdır -- request yapmak ve response almak için küçük, köklü, metin tabanlı bir biçim, MCP'nin kendisiyle ilgisi yoktur (MCP onu yalnızca kendi mesaj biçimi olarak benimsemiştir). Bir request bir **method** adlandırır ve **params** sağlar; bir response eşleşen bir **id** taşır ve ya bir **result** ya da bir hata içerir. "MCP'ye Giriş"in bir client'ın bir server'dan tool'larını listelemesini istemesini anlattığı yerde, o request ve response telde şöyle görünür:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "get_capital_city",
    "arguments": { "country": "Japan" }
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "content": [
      { "type": "text", "text": "The capital of Japan is Tokyo." }
    ]
  }
}
```

Bu ders için önemli olan nokta, MCP'nin kendi mesaj biçimini icat etmediği, JSON-RPC 2.0'ı wire format olarak kullandığıdır -- `tools/list`, `tools/call`, `resources/read` gibi işlemlerin çoğu bu request/response kalıbını izler (spesifikasyondaki bazı mesaj türleri, örneğin bir yanıt beklemeyen tek yönlü *notification*'lar, bu iki kutunun dışında kalabilir, ama bu ders seviyesinde onların ayrıntısına girmiyoruz). Bu, "TypeScript ile MCP Sunucusu Oluşturma"nın SDK kodunun sıradan fonksiyon call'larının arkasına gizlediği katmandır -- ama altta gerçekten gönderilen şey budur.

Transport'lar: stdio ve Streamable HTTP

JSON-RPC mesajlarının üzerinde seyahat edecek gerçek bir kanala ihtiyacı vardır -- MCP buna **transport** der, ve bu bilinçli olarak üstündeki her şeyden ayrı tutulur, böylece aynı server mantığı hangi transport onu taşırsa taşısın çalışır. En yaygın ikisi: **stdio**, client'ın server'ı yerel bir subprocess olarak başlattığı ve mesajların standart input/output'u üzerinden gittiği -- aynı makinede çalışan bir tool'la konuşan bir masaüstü AI uygulaması ya da IDE için tipiktir -- ve **Streamable HTTP**, server'ın bağımsız, muhtemelen uzak bir process olarak HTTP üzerinden ulaşılabilir çalıştığı, tek bir server'ın aynı anda birden fazla client ya da host tarafından paylaşılmasına izin veren. Hangi transport'un kullanıldığı, tool-calling mantığının kendisi için görünmezdir -- bir tool'un `name`'i, `description`'ı ve davranışı her iki durumda da aynıdır.

Bağlantı Yaşam Döngüsü: Initialize, Discover, Invoke

Aşağıdaki, MCP protokolünün olası bütün mesaj türlerinin eksiksiz bir dökümü değil -- bu derste tool/resource kullanımını anlamak için yeterli, bilinçli olarak sadeleştirilmiş bir temel yaşam döngüsü. Transport'tan bağımsız olarak, bir MCP bağlantısı bu üç aşamadan geçer:

1. **Initialize** -- başka bir şey olmadan önce, client ve server bir `initialize` request'i ve response'u değiş tokuş eder, bir protokol sürümü üzerinde anlaşır ve her taraf neyi desteklediğini belirtir (aşağıdaki "Capability Negotiation"a bakın).
2. **Discover** -- client neyin mevcut olduğunu sorar: tool'lar için `tools/list`, resource'lar için `resources/list`, prompt'lar için `prompts/list`, "MCP'ye Giriş"te kavramsal olarak tanıtıldığı gibi. Server, her öğenin adıyla, açıklamasıyla ve schema'sıyla yanıt verir.
3. **Invoke** -- client bir şeyi gerçekten kullanmak için bir request gönderir -- bir tool'u çalıştırmak için `tools/call`, bir resource'u getirmek için `resources/read` -- ve server

gerçek işi yapıp sonucu döndürür, tam olarak "MCP ve Tool-Calling Döngüsü"nün kavramsal olarak anlattığı gibi.

Tek bir bağlantı tipik olarak bir kez initialize aşamasından geçer, sonra kendi ömrü boyunca discovery ve invocation'ı defalarca tekrarlar.

Capability Negotiation

Initialize sırasında, client ve server yalnızca bir protokol sürümü üzerinde anlaşmaz -- her biri hangi opsiyonel özellikleri gerçekten desteklediğini de belirtir (örneğin, bir server'ın resource'ları hiç destekleyip desteklemediği, ya da bir client'ın belirli türde notification'lar alıp alamayacağı). Buna **capability negotiation** denir, ve var olma nedeni her host ya da server'ın her MCP özelliğine ihtiyaç duymamasıdır: yalnızca tool sunan minimal bir server, resource desteğini implemente etmek zorunda değildir, ve bir client yalnızca belirli bir server'ın gerçekten deklare ettiği yeteneklere hazırlanmak zorundadır, protokolün teorik olarak destekleyebileceği her yeteneğe değil.

Uygulamalı Örneğimiz Nereye Oturuyor (In-Memory Transport)

"TypeScript ile MCP Sunucusu Oluşturma", resmi TypeScript SDK'sını kullanarak gerçek bir client ve server inşa eder, ve bunları bir **in-memory transport** ile bağlar -- her iki taraf da aynı process içinde çalışır, yukarıda anlatılan aynı JSON-RPC mesajlarını değiş tokuş eder, sadece aralarında stdio ya da bir network socket'i olmadan. Bu, kendi kendine yeten, çalıştırılabilir bir ders için bilinçli bir sadeleştirmedir: gerçek bir kurulum neredeyse her zaman stdio ya da Streamable HTTP kullanır, server gerçekten ayrı bir process olarak. Initialize/discover/invoke yaşam döngüsü, mesaj biçimleri, ve tool-calling davranışı her iki durumda da aynıdır -- yalnızca altındaki transport değişir.

Best Practices

- MCP davranışı hakkında akıl yürütürken belirli bir transport varsaymayın -- "Transport'lar: stdio ve Streamable HTTP"nin gösterdiği gibi, bağlantı yerel (stdio) ya da uzak (Streamable HTTP) olsun, aynı server mantığı ve mesaj biçimleri geçerlidir.
- Bir bağlantı beklenmedik davranınca, hangi yaşam döngüsü aşamasında olduğunu kontrol edin (bkz. "Bağlantı Yaşam Döngüsü: Initialize, Discover, Invoke") -- `initialize` sırasındaki bir başarısızlık, bir `tools/call` invocation'ı sırasındakinden çok farklı bir sorundur.
- Bir server'ın her MCP özelliğini desteklediğini varsaymak yerine capability negotiation'a güvenin -- bir client, bir server'ı kullanmaya çalışmadan önce onun initialize sırasında gerçekten neyi deklare ettiğini kontrol etmelidir (bkz. "Capability Negotiation").

Yaygın Hatalar

- **JSON-RPC'nin MCP'nin icat ettiği bir şey olduğunu düşünmek.** "Kavramlardan Wire Format'a: JSON-RPC 2.0"ın açıkladığı gibi, JSON-RPC 2.0 önceden var olan, genel amaçlı bir mesaj biçimidir -- MCP yeni bir tane tasarlamak yerine onu benimsemiştir.
- **In-memory bir transport'un MCP'nin normalde nasıl deploy edildiği olduğunu varsaymak.** "Uygulamalı Örneğimiz Nereye Oturuyor (In-Memory Transport)" özellikle bu kursun örneğini kendi kendine yeten tutmak için seçildi -- gerçek deployment'lar neredeyse her zaman stdio ya da Streamable HTTP kullanır, server gerçekten ayrı bir process olarak.
- **Bir bağlantıyı hata ayıklarken doğrudan invocation'a atlamak.** "Bağlantı Yaşam Döngüsü: Initialize, Discover, Invoke"ın ortaya koyduğu gibi, initialize ve discovery önce gerçekleşir -- bir invocation hatasının gerçek nedeni genellikle bu önceki aşamalardan birindedir.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- MCP kendi mesaj biçimini icat etmez -- wire format olarak JSON-RPC 2.0'ı kullanır, önceden var olan, genel amaçlı bir biçim.
- Bir **transport** (yerel process'ler için yaygın olarak stdio, ya da uzak olanlar için Streamable HTTP) bu mesajları taşır; tool-calling davranışı hangisi kullanılırsa kullanılsın aynıdır.
- Her bağlantı aynı yaşam döngüsünden geçer: **initialize** (protokol sürümü ve yetenekler üzerinde anlaşma), **discover** (`tools/list` ve benzerleri), ve **invoke** (`tools/call` ve benzerleri).
- Initialize sırasındaki **capability negotiation**, her tarafın yalnızca gerçekten desteklediği opsiyonel özellikleri deklare etmesine izin verir.
- Bu kursun uygulamalı örneği, kendi kendine yeten kalmak için bir **in-memory transport** kullanır -- stdio ya da Streamable HTTP üzerinde olduğu gibi aynı yaşam döngüsü ve mesaj biçimleri geçerlidir.

Cheat Sheet

- Tel formatı = JSON-RPC 2.0 (request: method + params + id; response: id + result ya da error).
- Transport = stdio (yerel subprocess) ya da Streamable HTTP (uzak, paylaşılabilir).
- Yaşam döngüsü = initialize -> discover (`tools/list` , `resources/list` , `prompts/list`) -> invoke (`tools/call` , `resources/read`).
- Capability negotiation = her taraf, initialize sırasında desteklediği opsiyonel özellikleri deklare eder.

Terimler Sözlüğü

- **JSON-RPC 2.0:** MCP'nin client-server değiş tokuşları için wire format olarak kullandığı, önceden var olan, genel amaçlı, metin tabanlı mesaj biçimi.
- **Transport:** JSON-RPC mesajlarının üzerinde seyahat ettiği gerçek kanal -- yaygın olarak stdio (yerel subprocess) ya da Streamable HTTP (uzak).
- **stdio transport:** client'ın server'ı yerel bir subprocess olarak başlattığı ve mesajların standart input/output üzerinden gittiği bir transport.
- **Streamable HTTP transport:** server'ın HTTP üzerinden ulaşılabilir bağımsız bir process olarak çalıştığı bir transport.
- **Initialize aşaması:** herhangi bir MCP bağlantısının, client ve server'ın bir protokol sürümü üzerinde anlaştığı ve desteklenen yetenekleri deklare ettiği ilk aşaması.
- **Capability negotiation:** her tarafın, initialize sırasında hangi opsiyonel MCP özelliklerini gerçekten desteklediğini deklare etmesi.
- **In-memory transport:** client ve server'ın aynı process içinde çalıştığı, bu kursta uygulamalı örneği kendi kendine yeten tutmak için kullanılan bir transport.