Beyond JSON-RPC: Scaling Model Context Protocols with gRPC in Pytorch Ecosystem

Ashesh Vidyut, Software Engineer, Google
Madhav Bissa, Software Engineer, Google

MCP: Standardizing the AI Integration Landscape

The "N x M" Problem

Before MCP, developers faced a many-to-many integration nightmare:

- Tightly coupled custom code
- Rewrite needed for every tool
- Fragile data pipelines

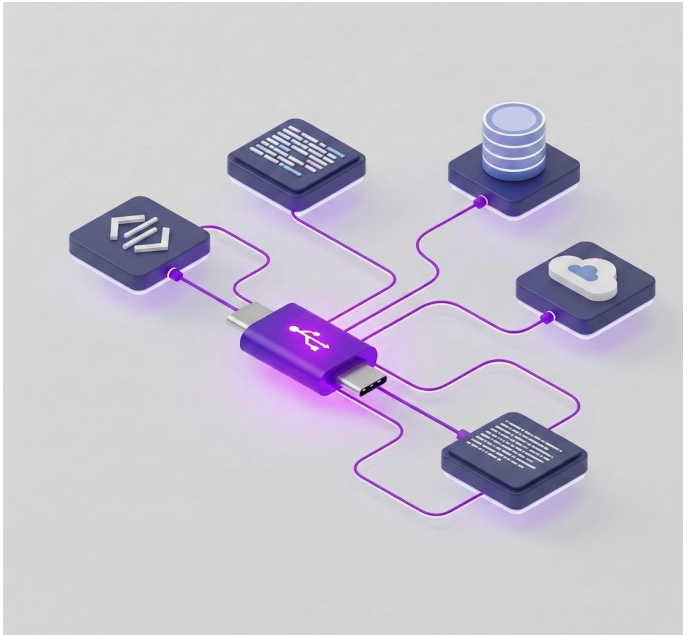
Standard Components

A client-server architecture replaces chaos:

- **Hosts:** AI Apps (IDE/Chat)
- **Clients:** Protocol connectors
- **Servers:** Resource bridges

Three Core Capabilities

Resources: Read-only data access (Docs, DB Schemas).
Tools: Executable functions (SQL, GitHub commits).
Prompts: Reusable workflow templates.



The "USB-C Port" for AI Integrations

The Current State: MCP Today

- HTTP and STDIO transports have proven highly successful for local scripts and simple agent implementations.
- Standardized JSON-RPC 2.0 messages enable rapid "Build once, integrate everywhere" development for many AI tools.
- Strong ecosystem growth with initial support for Claude, IDEs like Cursor, and various data connectors.

The Performance Wall

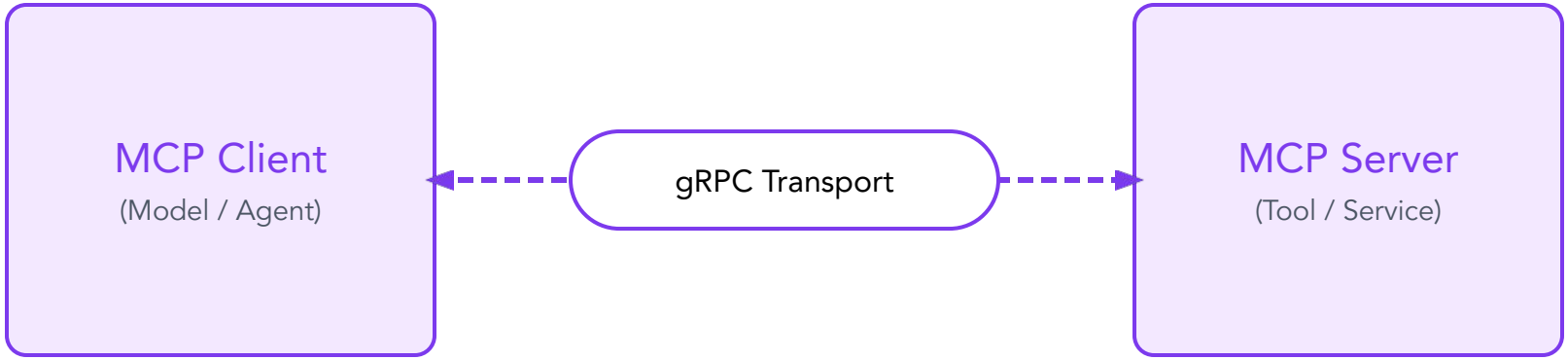
The Problem: Why JSON-RPC over HTTP struggles with production PyTorch

- **Serialization/Deserialization overhead:** Bulky JSON parsing consumes valuable CPU cycles and adds latency.
- **Moving large context windows:** Transmitting tensor metadata or massive contexts in text format leads to significant bandwidth bottlenecks.
- **The "Chatty" nature of tool calling:** Frequent requests add overhead milliseconds that accumulate into perceptible delays in real-time production environments.

The Vision

- The Core Idea: Bringing gRPC + Protobuf into the MCP ecosystem as a pluggable transport.
- The Goal: Minimal latency, maximum throughput, and production-grade reliability for the PyTorch ecosystem.

Architecture: MCP over gRPC



Spec Mapping: MCP Primitives gRPC Service Definitions

Resources: rpc ListResources / ReadResource | **Prompts:** rpc ListPrompts / GetPrompt | **Tools:** rpc ListTools / CallTool

Production-Ready: Auth & Security

Why gRPC is a "natural fit" for existing infrastructure.

- Native support for mTLS (Zero Trust environments).
- Integration with Service Meshes (Istio, Linkerd).
- Built-in Load Balancing and Health Checks.

Migration HTTP to gRPC: Server Side

HTTP / SSE

```
import torch
import uvicorn
from mcp.server.fastmcp import FastMCP


@fastmcp_server.tool()
def create_pytorch_tensor(dims: list[int]) ->
str:
  try:
    tensor = torch.rand(dims)
    return str(tensor)
  except Exception as e:
    return f"Error creating tensor: {e}"
app = fastmcp_server.create_starlette_app()

uvicorn.run(app, host="127.0.0.1", port=8080)
```

gRPC

```
import torch
import asyncio
from mcp_grpc_transport.server.grpc_server import
GRPCTransportSettings, serve_grpc


@fastmcp_server.tool()
def create_pytorch_tensor(dims: list[int]) -> str:
  try:
    tensor = torch.rand(dims)
    return str(tensor)
  except Exception as e:
    return f"Error creating tensor: {e}"
async def serve():
  target = 'localhost:50051'
  settings =
  GRPCTransportSettings(enable_reflection=True)
  await serve_grpc(fastmcp_server, target,
  settings)


asyncio.run(serve())
```

Migration HTTP to gRPC: Client Side

HTTP / SSE Client

```
import asyncio
import mcp
from mcp.client import sse

async def run_sse_client():
  mcp_server_url = "http://localhost:8080/sse"
  try:
    async with sse.sse_client(mcp_server_url) as
    streams:
      async with mcp.ClientSession(streams[0],
      streams[1]) as session:
        await session.initialize()
        tools = await session.list_tools()
        result = await
        session.call_tool('simple_tool', {'name':
        'SSE'})
        pytorch_result = await
        session.call_tool('create_pytorch_tensor'
        , {'dims': [2, 3]})
  except Exception as e:
    print(f'An error occurred: {e}')
asyncio.run(run_sse_client())
```

gRPC Client

```
import asyncio
from mcp_grpc_transport.client import
grpc_client_session

async def run_grpc_client():
  port = 50051
  target = f'localhost:{port}'
  session =
  grpc_client_session.GRPCClientSession(target=t
  arget)
  try:
    tools = await session.list_tools()
    result = await
    session.call_tool('simple_tool', {'name':
    'gRPC'})
    pytorch_result = await
    session.call_tool('create_pytorch_tensor',
    {'dims': [2, 3]})
  except Exception as e:
    print(f'An error occurred: {e}')
  finally:
    await session.close()
asyncio.run(run_grpc_client())
```

Key advantages of using gRPC Transport

- High performance & efficiency
- Simplified development
- Scalable and Reliable
- Native AI Security and Observability
- Flexible and Interoperable

What's Next (Roadmap and Ecosystem)

- Pluggable Transport Discussion with MCP SDK Maintainers.
- New MCP C++ sdk (with gRPC transport) in development.
- Other languages sdk like Go and Java with pluggable gRPC transport to follow.
- Blog Post - https://cloud.google.com/blog/products/networking/grpc-as-a-native-transport-for-mcp

What's Next (Repositories & Tools)

- gRPC Transport MCP Proto Repo - https://github.com/GoogleCloudPlatform/mcp-grpc-transport-proto
- gRPC Transport Pluggable Repo for Python - *(coming soon)* https://github.com/GoogleCloudPlatform/mcp-grpc-transport-py

Thank You