ChatGPT

OCPP 2.0.1 Gateway Middleware – Requirements and Design Overview

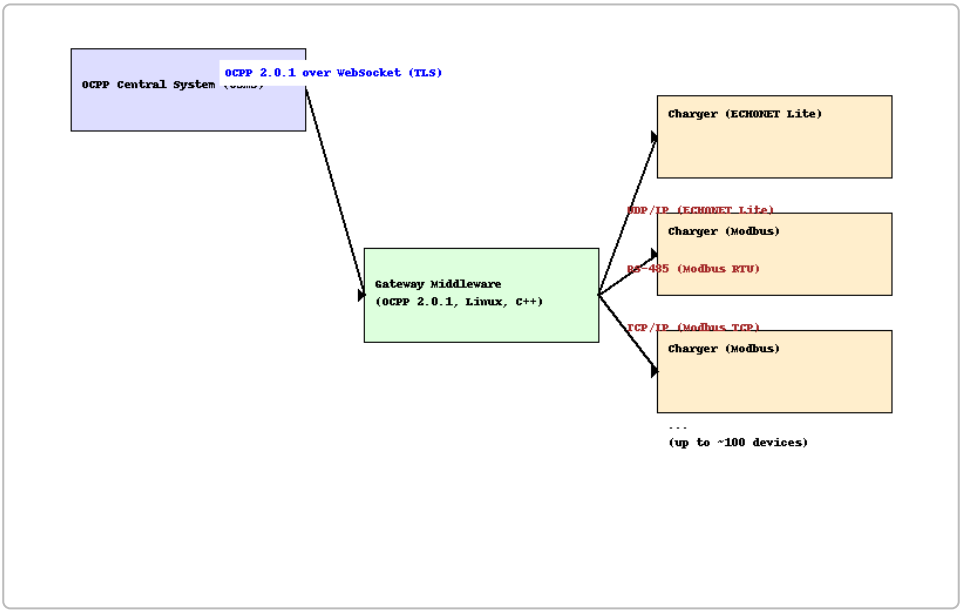

Figure: High-level architecture of the proposed OCPP 2.0.1 gateway middleware. The gateway (C++ service on Linux) bridges between an OCPP 2.0.1 Central System (over secure WebSocket) and multiple legacy EV chargers using ECHONET Lite (UDP/IP) and Modbus (RTU over RS-485 or Modbus/TCP over IP). It can support ~100 chargers concurrently (represented by the multiple device blocks).

Requirements Summary

Functional Requirements:

- **Protocol Translation:** The gateway must connect existing EV chargers (which communicate via ECHONET Lite or Modbus RTU/TCP) to an OCPP 2.0.1 network. It acts as a protocol translator, converting between OCPP 2.0.1 messages and the charger's native protocol commands. This allows non-OCPP legacy chargers to be integrated with a standard OCPP Central System.

- **OCPP 2.0.1 Compliance:** Implement an **OCPP 2.0.1 charge point client** stack on the gateway, supporting the core OCPP functionalities (registration, status, metering, transaction handling, etc.) and additional profiles as needed. The gateway should prioritize the **most widely-used OCPP 2.0.1 features** (see feature matrix below) to maximize interoperability [1]. For example, essential operations like authentication/authorization, start/stop charging sessions, heartbeats, status notifications, and meter readings are critical [2]. Optional OCPP features (e.g. reservations, local white-list, smart charging schedules, firmware updates) should be implemented with priority based on industry use frequency (detailed in the next section).

- **Flexible Mapping Configuration:** Provide a **flexible mapping mechanism** between OCPP data elements and the charger's native protocol parameters. The gateway should allow **mapping**

definitions (e.g. which Modbus register or ECHONET property corresponds to a given OCPP variable or event) to be easily configured **without code changes**. This could be done via external configuration files (e.g. JSON/YAML defining mappings), a GUI-based management tool, or other admin interfaces. The mapping system should accommodate differences in charger models – for instance, allow using templates or profiles per device type, which can be imported or reused for quick setup [3] [4]. This flexibility ensures that new charger models or changes in protocol data points can be handled by updating configurations rather than altering program logic.

- **Concurrent Device Connections:** Support on the order of **100 chargers connected concurrently** through the gateway. The architecture must handle multiple simultaneous communications: up to 100 OCPP client sessions (one per charger or per connector) with the central system, and connections to up to 100 field devices (via ECHONET Lite or Modbus). The system should maintain performance and reliability at this scale – for example, non-blocking, asynchronous I/O or multithreaded design may be needed so that polling one device or waiting on one network operation does not block others. It should also manage resource usage for 100 WebSocket connections (if treating each charger as an independent OCPP endpoint).

- **Scalability and Modular Design:** The software should be structured in a modular way, isolating the OCPP interface, the protocol driver for ECHONET, and the protocol driver for Modbus, with a clearly-defined mapping layer in between. This will make it easier to maintain and extend. For example, the gateway might contain:

- An **OCPP Client Module** (manages WebSocket connections to CSMS, handles OCPP messaging for each charge point).
- A **Protocol Adapter Module** for ECHONET Lite (handles UDP communications, device discovery, property read/write).
- A **Protocol Adapter Module** for Modbus (possibly supporting both RTU and TCP transports).
- A **Mapping/Translation Engine** that translates OCPP calls to one or more operations on the device protocols and vice versa (using the mappings defined in configuration).

- Optionally, a **Management Interface** (could be a CLI, web UI, or config files) to configure devices, mappings, and monitor status.

- **Linux Deployment:** The gateway middleware must run on a **Linux platform**. It should operate as a background service/daemon on Linux, and leverage OS capabilities (e.g. serial ports for RS-485, network stack for TCP/UDP). The design should be portable across common Linux distributions for embedded or server systems. Using standard C++ libraries (and possibly existing open-source libraries for OCPP, Modbus, or ECHONET) is encouraged to speed development and ensure reliability on Linux.

- **Security Requirements:** Ensure secure communication on the OCPP side – OCPP 2.0.1 typically uses **WebSocket over TLS** for communication with the central system, so the gateway must handle certificate-based TLS connections. If mutual authentication or OCPP security profiles are required by the CSMS (e.g. client-side certificates in **Advanced Security** profile), the gateway should support those in priority or plan to add them [1]. On the device side, ECHONET Lite and Modbus are generally in a local network; however, measures like network segmentation or VPN may be considered since those protocols may not have built-in encryption. The gateway should also enforce role-based access if it has a management UI, and securely store any sensitive config (like keys or passwords for devices).

- **Maintenance & Extensibility:** The solution should be maintainable and extensible. This includes:

- Providing logging and diagnostics for communications (to troubleshoot mapping issues or device communication problems).
- The ability to update mapping configurations or add new device drivers with minimal downtime.
- If feasible, support remote software updates of the gateway itself (since OCPP has a firmware update feature, possibly this could be leveraged to update the gateway's software if the central system triggers an update – effectively treating the gateway as the "charging station" for firmware management purposes).
- Consideration for future protocols or extensions (e.g. if later needing to support OCPP 2.1 or other charger protocols, the architecture should accommodate adding new modules).

Performance Requirements:

- The gateway should introduce minimal latency in translating commands. For example, when the central system sends a Remote Start Transaction, the gateway should promptly relay the equivalent command to the charger (via Modbus or ECHONET) and respond. Round-trip times should be low (a few hundred milliseconds ideally, not counting any inherent network delay).
- It must reliably handle the throughput of periodic messages. With ~100 chargers, if each sends regular **MeterValues** and **StatusNotifications** (as per OCPP), the system could be processing dozens of messages per second. The design should queue and process these efficiently (possibly with asynchronous message handling and worker threads). Support for WebSocket compression (an OCPP 2.0.1 feature) can help reduce bandwidth if needed.
- For Modbus RTU connections, consider the polling frequency and serial bus speed: With many devices on one RS-485 line, the polling interval might need to be tuned so as not to overload the bus. The system may need to manage staggered polling or event-triggered reads (if the charger can signal when to read data) to scale to many devices.

Constraints:

- **Protocol Limitations:** ECHONET Lite and Modbus have different data models from OCPP. The design must accommodate the fact that not every OCPP feature may cleanly map to a given device. For instance, a legacy charger might not support reservation or may not measure energy with high precision. The requirement is to handle such gaps gracefully – e.g. by reporting "Not Supported" for certain OCPP operations if the underlying device can't do it, or by emulating where possible. Only the OCPP features that are **most common and feasible** should be initially implemented; other features can be stubbed or left for future (see prioritization below).

- **Operating Environment:** The gateway will run on Linux – likely on an industrial PC or embedded system in the field (potentially installed on-premises where the chargers are). It should be robust to run 24/7 and recover from errors (e.g. lost device communications or network glitches). It should also handle reconnect logic: if a charger goes offline or a connection is lost, the gateway should detect it and notify the CSMS via OCPP status, and attempt reconnection if appropriate. Similarly, if the central system connection drops, the gateway should keep the charger sessions running offline and sync when reconnected.

- **ECHONET Lite Specifics:** ECHONET Lite typically uses UDP/IP (often on a well-known port 3610) for device communication. The gateway might need to send multicast discovery messages to find ECHONET devices, or have pre-configured IP addresses. It must implement the ECHONET Lite protocol to get/set specific device object properties. Ensuring reliability over UDP (which is connectionless) is important – for example, the gateway may need to resend a request if no response is received, and handle out-of-order or lost packets. This is an important consideration for maintainability (discussed further in the communication methods section).

- **Modbus Specifics:** For Modbus RTU (serial over RS-485), the gateway needs access to serial ports (with appropriate driver settings for baud rate, parity, etc.). Timing is critical on RS-485; the software must insert proper delays between frames and manage the DE (Driver Enable) line if using half-duplex transceivers. Using a well-tested Modbus library or stack is advisable to avoid low-level bugs. For Modbus TCP, the gateway will act as a client connecting to charger's Modbus servers (or to serial-to-TCP converter endpoints). It must manage socket connections and potentially multiple concurrent requests. The design should consider using non-blocking sockets or a connection pool for efficiency.

- **Administration:** The system should provide an interface for administrators to:

- Configure new chargers (e.g. add a device with certain protocol type and address, apply a mapping template for it).
- Adjust mapping definitions if needed.
- Monitor the status of each connected charger (e.g. online/offline, last heartbeat time, last meter reading, etc.).
- View logs or diagnostic info for troubleshooting.

With these requirements in mind, we now detail the architecture and design approach.

Recommended Architecture

The gateway is conceived as a **middleware service** that sits between the Central System (CSMS) and the field devices (chargers), as illustrated above. The **overall architecture** can be described in layers and components:

- **Upstream Interface (OCPP Client):** The gateway implements the OCPP 2.0.1 protocol on behalf of the chargers. This could mean maintaining **multiple OCPP identities** – one per physical charger – each appearing as an individual "Charge Point" to the central system. In practice, this means the gateway might spawn 100 separate OCPP WebSocket client connections to the CSMS (one for each charger it manages), each with its own ChargeBox ID or credentials. This approach keeps each charger logically separate for the back-end. (An alternative design is to use OCPP 2.0.1's support for multiple connectors under one charge station identity. In that case, the gateway could present itself as one charging station device with 100 connectors/EVSEs. This may simplify the number of connections, but it complicates mapping and might be less flexible if chargers are distributed. Most implementations choose one OCPP connection per physical charger for clarity.) The OCPP client layer should handle all message exchange with the CSMS: BootNotification on startup, periodic Heartbeats, StatusNotification for each connector/EVSE, Authorize/StartTransaction/StopTransaction when a session begins or ends, data transfer messages, etc.

- Concurrency: To handle many OCPP connections simultaneously, an **event-driven asynchronous framework** is recommended. This could be achieved with tools like Boost.Asio or using an asynchronous WebSocket library. By using a single I/O service with multiple strands or a pool of threads, the gateway can manage many socket connections efficiently. A persistent WebSocket connection is maintained for each charge point to avoid repeated handshakes and reduce overhead, as OCPP relies on long-lived connections for realtime control [5]. Each connection's incoming messages should be handled in a non-blocking way – for example, by enqueueing requests from the CSMS and processing them without stalling the network reads. Similarly, sending data (like meter values) for each device should not interfere with others (this can be achieved by per-connection queues and possibly a thread pool consuming those queues).

- OCPP Library: It may be beneficial to use or adapt an existing OCPP 2.0.1 library or reference implementation if available in C++. (The Open Charge Alliance has open-source efforts like **MicroOCPP/OpenOCPP** for lightweight clients [6], and other vendors have SDKs.) This would ensure compliance with the 1500-page spec while saving development time. The library would provide encoding/decoding of OCPP JSON messages, schema validation, and perhaps handle the device model structure. If no suitable library, a custom implementation must carefully implement the core profiles and handle JSON WebSocket communication, perhaps by following the OCPP 2.0.1 specification and leveraging JSON schema or code generation for messages.

- **Device Communication Layer:** On the downstream side, the gateway includes **drivers or adapters** for each protocol:

- **ECHONET Lite Adapter:** Likely implemented as a UDP client/server. The gateway may need to join a multicast group or send a broadcast to discover devices initially. For direct control, it will send UDP packets to the device's IP (each ECHONET Lite device has an IP address on the LAN) with the appropriate ECHONET Lite frames (including the ECHONET object class, instance, and property codes to get or set). It will then listen for response packets. This adapter should handle ECHONET-specific logic such as:

    - Building and parsing the ECHONET Lite message format (which includes an EHD header and data section with property values).
    - Knowing the specific device object for an EV charger. (For example, an EV charger might implement the "EV Charging Device" object in ECHONET, and have properties like operational status, charging current, etc. The adapter should use the correct EPC codes to read those values or issue controls like start/stop charging.)
    - Possibly maintaining a socket open to receive **notifications** if the device sends spontaneous reports. (ECHONET Lite allows devices to send notifications when certain values change. If the chargers use that, the gateway can take advantage so it doesn't have to poll continuously.)
    - Retrying requests if no response is received (to account for UDP packet loss or device busy).
    - Handling multiple devices: this adapter could use a single UDP socket to communicate with all ECHONET devices (since UDP is connectionless, it can send to various addresses). It must differentiate responses by source address and ECHONET object ID. This might be done in an async loop where each incoming packet is identified and routed to the corresponding device context.

- **Modbus Adapter:** There are two sub-modes:

    - Modbus RTU: The gateway should interface with an RS-485 line (through a serial port). All chargers on that line are typically daisy-chained, each with a unique Modbus ID. The adapter must periodically poll each device's registers or coils for data (like charging status, meter values) and issue commands (like enabling output, etc.) when instructed. Because RS-485 is half-duplex and all devices share one medium, polling must be serialized. The adapter can iterate through a list of device IDs, sending a request and waiting for the response (with a timeout). If 100 devices were on one line, this could be slow; in practice, it may be better to use **multiple RS-485 buses** (with multiple serial ports) to split the load (e.g. 4 buses with 25 devices each) or use **Modbus/TCP gateways** to parallelize (discussed in Communication section). The adapter should manage low-level details: ensuring the interval between frames is per spec, configuring baud rate (likely 19200 or 9600 on many chargers, but could be higher), and handling error responses or absent devices (mark

device as offline if no response). A robust **Modbus library** (like libmodbus in C) could be utilized for parsing and frame formation to reduce errors.

- Modbus TCP: For devices that have an Ethernet interface or are connected via a serial-to-TCP converter, the adapter will maintain a TCP/IP connection to each device's Modbus port (usually 502). Modbus TCP allows concurrent handling since each device is independent – the adapter could either keep one socket per device open or open on-demand. Keeping them open is fine for 100 devices, as 100 idle TCP connections are not a big load for a Linux system. The adapter can send requests in parallel (especially if using non-blocking sockets or a thread per connection) since each device is separate. This yields better performance than a single RS-485 bus. Timeout and error handling are still needed (if a device doesn't respond or the connection drops, mark it offline and attempt reconnection periodically).

- Both adapters will feed data to the mapping/translation layer and execute commands given by the mapping layer. To maximize throughput, the design could use **producer-consumer patterns**: e.g., an OCPP command arrives (like RemoteStop for Charger X) – the mapping layer produces a task for the Modbus adapter (e.g. "write value to register Y of device X"), the Modbus adapter thread picks it up and executes it. Conversely, the Modbus adapter upon polling can produce an event (e.g. "device X register Z changed to value V") which the mapping layer consumes and turns into an OCPP message (like a StatusNotification or MeterValues for that charger).

- **Mapping & Translation Layer:** This is the core of the gateway's logic – it **bridges the semantic gap** between OCPP and the device protocols. The mapping layer should be driven by configuration data as much as possible:

- It maintains a **data model or mapping table** that for each charger and each relevant parameter, defines how to get or set that parameter on the device. For example, it might know that for Charger #5 (type = "ModbusVendorA"), the EVSE status is indicated by Modbus register 100 (1 = available, 2 = occupied, etc.), and the total energy meter is at register 110 (32-bit value), etc. Similarly, for Charger #8 (which is an ECHONET device), it knows which EPC codes correspond to, say, "Charging in progress" status and energy delivered.
- The mapping config can also define **trigger conditions**: e.g., if a certain Modbus coil turns from 0 to 1, that means a vehicle just plugged in – so the gateway should emit an `StatusNotification` (Plugged In) followed by perhaps an `Authorize` request flow. This layer thus might contain state machines or logic for certain sequences. However, ideally even such logic can be data-driven or minimal, delegating to OCPP's rules as much as possible.
- The gateway likely needs to implement an internal representation of each charger's state (to track whether a transaction is active, etc.). OCPP 2.0.1 introduced a **Device Model** which represents the station's components and variables [7] [8]. The mapping layer essentially populates and updates this device model for each charger by reading the field data. For initial simplicity, the gateway can manage a few key states: Available/Occupied/Faulted status, Authorized tag for current session, Energy meter readings, etc., which are enough for basic charging sessions.
- **Configuration Management:** The mapping definitions should be stored in a structured configuration (e.g. an XML, JSON, or database). For instance, a JSON snippet might define a charger model mapping as:

```
"VendorX_ChargerModel123": {
  "availability_status": { "modbus_register": 100, "datatype": "uint16", "mapping": { "0":
"Available", "1": "Occupied", "2": "Faulted" } },
  "active_power":     { "echonet_eoj": "0x02:0x88:0x01", "epc": 0xE0 },
  "start_charge_cmd":   { "modbus_coil": 5, "write_value": 1 }
```

```
  /* ... etc ... */
}
```

This is a conceptual example showing how a mapping might be specified (with a Modbus register mapping and an ECHONET mapping). The gateway at runtime would load such mappings. This allows **flexibility** to accommodate various devices. An admin could edit the config or select a different template for a new charger type. (In practice, one can also present this in a GUI form for easier management.)

- The mapping layer performs transformations: e.g., converting raw values to engineering units or OCPP-defined enumerations. If a Modbus register gives a value in tenths of ampere, the mapping can scale it to amps before sending in an OCPP MeterValue. It also could accumulate values if needed (like summing phase currents, etc., depending on OCPP requirements).
- **Prioritization:** The mapping should be designed to handle the **most critical interactions first** – such as session authorization and start/stop. For example, when an EV driver presents an RFID (which might be detected by the charger and put into a certain register or ECHONET property), the gateway must capture that event and translate it to an OCPP `Authorize` and possibly `StartTransaction` sequence. Similarly, if the CSMS sends a `RemoteStopTransaction` via OCPP, the mapping needs to know how to command the charger to stop (maybe by writing a coil to open a contactor or sending an ECHONET "stop charging" property). These real-time controls are top priority. Less critical data (like updating a display message or reading a temperature) can be lower priority in implementation.

- **Example from Existing Gateway:** Notably, an existing OCPP-to-Modbus gateway product supports only a core set of operations and uses a **fixed register map** approach [9] [2]. In that device, specific Modbus registers correspond to OCPP commands like Authorize, Start Transaction, etc., and the integrator uses a PC tool to link those to their PLC or BMS. Our gateway will use a more flexible mapping, but the principle is similar: define each OCPP action/data and how it ties to the charger's interface.

- **Internal Data Flow:** Putting it all together, a typical flow might be:

- **Boot/Initialization:** On startup, the gateway loads configuration (including a list of chargers to manage and their protocol types and addresses, plus mapping templates for each). It then opens connections: e.g. establishes WebSocket to CSMS for each charger (sending BootNotification for each) and opens necessary device comm channels (opening serial ports, binding UDP socket, etc.). Each charger is set to "Unavailable" state until communications are confirmed.
- **Heartbeat & Status Loop:** The gateway sends periodic Heartbeats to the CSMS for each OCPP connection as required. It polls each device's status via Modbus/ECHONET on a schedule. When status changes (e.g. a car plugged in), the mapping layer detects the change and triggers an OCPP `StatusNotification` to inform the CSMS. The status might change from "Available" to "Occupied" (EVSE is in use but not charging yet) or "Charging" (once current flows). These status updates are crucial for the network operator to know charger availability.
- **Authorize and Start Transaction:** If the charger itself reads an RFID card or a PIN (in local use), the gateway might need to capture that and forward to CSMS for authorization. In a Modbus context, this could mean the charger's controller writes the presented tag ID into some registers. The gateway mapping layer would notice a non-zero value in "TagID register", and then call the OCPP `Authorize` message with that ID. If the CSMS approves, it might send a `SendLocalStartTransaction` or simply the charger will start if it was pre-configured. Alternatively, the system may rely on **Remote Start**: The user uses a mobile app that triggers a remote start from CSMS. In that case, the gateway receives `RemoteStartTransaction` from OCPP. The mapping layer then issues the command to the charger (e.g. set "start charging" coil = 1). The charger

begins charging, and the gateway then sends a `TransactionEvent` / `StartTransaction` to the CSMS to record the session start.

- **Metering:** During a charge, the gateway periodically reads the meter values from the charger (e.g. energy in kWh, current, voltage). According to OCPP settings, it might send a `MeterValues` report every X minutes or at transaction end. Accurate energy reporting is important for billing, so the mapping must ensure the units and format are correct (e.g. OCPP typically expects Wh or kWh readings, with a timestamp).
- **Stop Transaction:** When the user stops charging (unplugs or presses stop), the charger will signal this (maybe a bit goes low or the current drops to 0). The gateway detects it and sends a `TransactionEvent` (Stop) or `StopTransaction` to CSMS with the final readings. Conversely, if a remote stop was requested by the CSMS (e.g. user commands via app), the gateway receives `RemoteStopTransaction`, then it issues the appropriate command to cease charging on the device. It should also ensure the charger physically stops (perhaps by waiting for confirmation or by reading current = 0), and then it confirms by sending the StopTransaction to CSMS.

- **Handling Multiple Devices:** All the above happens concurrently for up to 100 chargers. Thus, the architecture likely uses **multi-threading or asynchronous tasks**. For example, one thread (or async task) could handle all OCPP messaging, while another handles polling devices, but better is to have them decoupled per device: each charger could be managed by an independent state machine or set of tasks. This isolation prevents a slow device from backlogging others. Given the scale, the solution might employ a small thread pool for device communications and a message queue system to communicate between the OCPP side and device side for each event, ensuring thread-safe operations.

- **Data Persistence:** The gateway may need to store certain data persistently:

- OCPP 2.0.1 requires some configuration to be reportable (like configuration keys, which could just be static mappings in code or config file). If implementing **Local Authorization List** (whitelist of RFID cards for offline use), the gateway should store and update that list (OCPP has messages to update the local list). This could be a file or small database on disk.
- If the gateway supports **Offline transactions** (i.e., if the CSMS is offline, it can still allow charging and later report the data), it must cache those transaction records safely until the connection is restored. So some kind of lightweight database or file storage for queued events might be needed to avoid data loss (especially for billing info).

- Logging of transactions or significant events for audit may also be required by the operator.

- **Fault Handling:** The architecture must handle error conditions robustly:

- If a charger is not responding (e.g., Modbus timeouts), mark it as unavailable and send OCPP status (Faulted or Unavailable) to the CSMS [10] . Perhaps attempt reconnection or notify maintenance.
- If the gateway loses network to CSMS, it should perhaps continue to allow charging (if configured to do so) using local auth if available, and queue any messages to send later. When network restores, send a BootNotification and replay any buffered events (OCPP 2.0.1 supports offline transaction queueing and reconciliation).
- If the gateway itself needs to restart (e.g., software update), design a strategy to minimize impact: ideally it should remember ongoing sessions so that after a reboot it can recover (or at least notify CSMS of a reboot and current status of connectors).

To summarize, the architecture follows a **gateway pattern**: one side speaks OCPP 2.0.1 to the cloud, the other side speaks device-specific protocols, with a flexible translation in between. It emphasizes

modularity (so protocol specifics are isolated), configurability (to support different device mappings), and concurrency (to handle many devices in parallel). The design also prioritizes the most commonly used functionality in EV charging scenarios to deliver value quickly. Next, we provide a matrix of OCPP 2.0.1 feature support prioritization, and then discuss the mapping design approach and communication methods in detail.

Prioritized OCPP 2.0.1 Feature Support

OCPP 2.0.1 comprises a large number of features grouped into functional profiles [1]. Not all need to be implemented initially – we should focus on the features that are most commonly required in real-world charging networks (ensuring basic interoperability and essential smart charging capabilities first). The table below lists the major OCPP 2.0.1 feature groups and their recommended priority for implementation:

OCPP 2.0.1 Feature Group	Description & Examples	Usage Frequency (Industry)	Priority
Core Functionality (Core Profile) [1]	Basic OCPP operations needed for any charger. Includes bootstrapping and configuration, authentication, session management, and status reporting. Examples: BootNotification, Heartbeat, Authorize/Authenticate user, StartTransaction/ StopTransaction (or TransactionEvent messages), StatusNotification (availability status), basic error reporting. Also basic **security** handshake (TLS and key/certificate provisioning as needed).	**100%** – Used by essentially all deployments. Without these, a charging station cannot function on OCPP [11].	**Must Implement (Highest)** – Implement in Phase 1. This is the minimum for OCPP compliance (Core Profile is required for certification) [12].
Remote Control Operations (part of Core & Remote Trigger)	Remote execution of commands from the CSMS. Examples: RemoteStartTransaction and RemoteStopTransaction (allow the backend to start/stop a session), UnlockConnector (to remotely release a stuck connector latch), and TriggerMessage (CSMS-triggered data reporting).	**Very High** – Remote start/stop is widely used for mobile app control of charging, and unlocking is important for support. Many networks rely on these for user convenience and operational control [2] (includes RemoteStart/Stop).	**Must Implement (High)** – Phase 1. These often fall under core or remote control profiles and are essential for CPO operations.

OCPP 2.0.1 Feature Group	Description & Examples	Usage Frequency (Industry)	Priority
Metering & Energy Reporting (Meter Values)	Periodic or event-based sending of meter data: power, energy, current, voltage, etc. Examples: `MeterValues` messages during a transaction, final energy delivered at session end, etc. OCPP 2.0.1 allows granular or composite reporting of these values.	**High** – Critical for billing and energy management in almost all scenarios [2] (MeterValues listed as supported operation). Operators need energy readings for every session.	**Must Implement (High)** – Phase 1. Ensure at least total energy and possibly current/voltage if needed for monitoring.
Smart Charging (Load Management) [10]	Adjusting charging power based on external commands or schedules. Examples: `SetChargingProfile` to limit current or schedule charging periods, `ClearChargingProfile`, `GetCompositeSchedule` for load planning. This also covers both per-transaction limits and overall station limits. In OCPP 2.0.1, smart charging is very rich, including the ability for the CSMS to control charging profiles for EV or EVSE, respond to grid or tariff signals, etc.	**High** – Load management is increasingly important for sites with multiple EVSEs or limited power availability. Many CPOs use it to prevent overloads and to integrate with energy tariffs. Smart charging (scheduling and power capping) is one of the key enhancements of OCPP 2.x [13] [14]. However, note that legacy chargers might only support simple implementations (e.g., a command to set max current).	**High Priority** – Phase 1 if possible, at least in a basic form. Implement the ability to receive a current limit from CSMS and apply it via Modbus/ ECHONET (if the charger supports adjusting current). Full support for schedules and profiles can be iterative, but the basic power limiting should be included early.

OCPP 2.0.1 Feature Group	Description & Examples	Usage Frequency (Industry)	Priority
Local Authorization List Management [15]	Handling a local whitelist of authorized RFID cards/badges in the charger for offline use. Allows CSMS to send a list of authorized IDs to the station. Examples: `SendLocalList`, `GetLocalListVersion`, and using local auth if CSMS is unreachable.	**Moderate** – Used in scenarios where network connectivity is unreliable or for redundancy. Many public networks rely on online auth, but some enterprise or private installations use local lists.	**Medium Priority** – Implement in Phase 2. Not critical if the environment has constant connectivity, but good to have for resilience. The gateway can maintain the list and perform local authorization when CSMS is offline [15].
Reservation (Reservation Profile) [16]	Reserve a charging station in advance for a specific EV driver. Examples: `ReserveNow` to reserve an EVSE for a period, `CancelReservation`.	**Variable (Low to Moderate)** – Some networks (especially in Europe or commercial parking) use reservations via mobile apps, but many others do not use this feature heavily yet. It's not universally implemented by all chargers.	**Medium Priority** – Phase 2. Nice-to-have if the operator plans to offer reservations. The gateway would need to handle a reservation state and block the charger locally when reserved. If time/ resources permit, include it; otherwise it can be added later without affecting core charging.

OCPP 2.0.1 Feature Group	Description & Examples	Usage Frequency (Industry)	Priority
Firmware Management [17] [18]	Remote firmware updates of the charging station. In OCPP 2.0.1, this involves the CSMS instructing the device to download and apply new firmware. Examples: `UpdateFirmware` request (with URL to firmware file), `FirmwareStatusNotification` (to report download and update progress).	**Medium** – Important for maintaining stations, but in our case the "station" is the legacy charger. If the charger does not support remote firmware update via its interface, this may be moot. However, this feature could be repurposed to update the gateway itself. Many OCPP implementations include firmware update to keep hardware up-to-date [18].	**Medium Priority** – Phase 2 or 3. If feasible, implement the calls so the gateway can respond properly (even if just to say "firmware updated" without actually doing anything on the charger). If treating the gateway as the device, it could apply to updating the gateway's software. Not critical for initial bring-up of charging functionality.
Advanced Device Management (Monitoring & Reporting) [19]	Enhanced monitoring of station variables and device model components, and custom reports. Examples: `GetVariables/SetVariables` to read or set configuration items, `GetReport/NotifyReport` for detailed device info, `SetMonitoring` to set threshold alarms on variables. OCPP 2.0.1 device model allows a rich set of data to be monitored (temperatures, internal faults, etc.).	**Medium** – Very useful for high-end chargers and proactive maintenance, but legacy devices may not expose many telemetry points. Many operators initially focus on basic operation and add monitoring later.	**Medium Priority** – Phase 2. Implement at least basic `GetVariables/SetVariables` for common configuration (like change charger availability, etc.). Custom monitoring or extensive reports can be added gradually. The gateway can focus on a subset of critical parameters (e.g., status, power) for monitoring in early versions.

OCPP 2.0.1 Feature Group	Description & Examples	Usage Frequency (Industry)	Priority
Display & User Interface Elements (Advanced UI Profile) [20]	Sending messages to display, showing pricing or text on the charger's screen, controlling LEDs, etc. Examples: `DisplayMessage` to show a text to the user, `SetChargingProfile` with a cost information, or `TariffCost` in transaction messages.	**Low** – Only applicable if chargers have a user display or need to show info like tariffs. Many basic chargers lack screens, or the functionality isn't commonly used yet in OCPP networks.	**Low Priority** – Phase 3. Implement only if needed. If the chargers do have a display and the operator wants to push messages (e.g. "Welcome" or pricing info), this can be supported. Otherwise, safe to defer.
Advanced Security (Security Profile) [21]	This involves client-side certificates, secure key exchange, etc., as defined in OCPP 2.0.1 security white paper. For example, support for installing a client certificate signed by a CA, using `CertificateSigned`, or secure communication with an embedded HSM.	**Medium** – Security is highly important, but many networks simplify by pre-loading certs or using basic TLS. However, OCPP 2.0.1's advanced security profile is gaining adoption for certificate management and secure provisioning (especially in large networks). E.g., EV Connect's OCPP 2.0.1 certification included Advanced Security [22].	**Medium Priority** – Implement foundational security (TLS) in Phase 1 (mandatory). Advanced features like certificate signing can be Phase 2 if required by the CSMS. Ensure the architecture can accommodate certificate handling; implement messages like `SignCertificate` if needed by the ecosystem [23].
ISO 15118 Integration (Plug and Charge) [24]	Support for exchanging EV certificates and enabling Plug-and-Charge (PnC) per ISO 15118-2/20. This includes certificate installation (`InstallCertificate`), handling EV Public Key Infrastructure, etc. Essentially allows EV and charger to do contract authentication without user input.	**Low (currently)** – This is cutting-edge and only applicable if the charger and EV both support Plug & Charge. For older chargers (likely not ISO 15118 capable), this is not applicable. Only new installations with ISO 15118 would need this, and it's still emerging in 2025.	**Low Priority** – Can be skipped in initial implementation. If future hardware upgrades support it, then consider adding. For now, focus on RFID/app authentication which is the norm for these legacy devices.

OCPP 2.0.1 Feature Group	Description & Examples	Usage Frequency (Industry)	Priority
Diagnostics (Logs retrieval) [25]	Uploading diagnostic information or crash logs from the charger to the CSMS. Example: `GetLog` request and `LogStatusNotification` to retrieve logs, or `GetDiagnostics` (in OCPP 1.6) equivalent.	**Low** – Rarely used on a routine basis; mostly for troubleshooting specific issues. Not all CSMS platforms even expose this to end users.	**Low Priority** – Phase 3. Implement if needed for maintenance. The gateway could potentially fetch internal logs or device error logs via Modbus and send them, but this is not essential for core functionality.
Custom / Proprietary Extensions (DataTransfer) [26]	The ability to send custom messages for non-standard functionality. `DataTransfer` in OCPP allows charger and CSMS to exchange vendor-specific info.	**Varies** – Sometimes used for firmware-specific commands or manufacturer telemetry outside the OCPP spec. Not generally used unless needed for a particular hardware.	**Low Priority** – Implement basic support (so the gateway can respond to a DataTransfer even if just acknowledging). This is straightforward to add, so can be done in Phase 1 or 2 with low effort.

Notes on Prioritization: The **Core profile and immediate operations** (authorization, transactions, status, basic remote control) are absolutely first priority [12] [2] . Without these, the system won't deliver any charging service. Right after core, **smart charging** and **metering** are the most significant features to include – as they impact energy management and billing which are key motivations for using OCPP. Features like reservations, local lists, and firmware update are secondary – useful in certain deployments but not universally required initially. They should be added once the primary functionality is stable. This phased approach aligns with how many manufacturers implement OCPP: start with core and a couple of optional profiles, then gradually certify more profiles over time [15] .

It's also worth noting that our gateway's ability to implement some features may depend on the underlying chargers' capabilities. For example, if a charger can't physically limit current, the Smart Charging profile cannot be fully realized – we might only implement a subset (e.g., the gateway could simulate acceptance of a profile but not actually enforce it, which is not ideal). In requirements, we should mark clearly which OCPP features are **supported vs. not supported** by the gateway given the connected hardware. This can be presented in documentation as a compliance matrix for the product.

Mapping Design and Configuration Strategy

One of the most important aspects of this middleware is the **mapping between OCPP and the charger protocols**. We need a design that is both **flexible** and **manageable**:

- **Configuration-Driven Mapping:** The mapping will be defined in configuration files or data structures, rather than hard-coded. This allows modifications without recompiling code. As mentioned, a JSON or YAML file (or a database) can list each device type's mapping of OCPP variables/operations to protocol specifics. For example:
- An OCPP `StatusNotification` could be triggered by either a Modbus register change or an ECHONET property indicating the connector's state. The config would specify which register/property and the logic to interpret it (e.g., 0 = Available, 1 = Occupied, 2 = Charging, 3 = Faulted).
- An OCPP `Authorize` request may not map to a device action (since the CSMS handles auth), but if the charger provides an RFID read event, that event mapping should call the OCPP Authorize.
- OCPP `StartTransaction` and `StopTransaction` messages are a bit unique in OCPP 2.0.1 (they are parts of a TransactionEvent flow). However, conceptually they mark the boundaries of a session. The mapping needs to align the charger's notion of a session with OCPP's. If the charger itself doesn't track sessions (some just operate on a simple on/off), the gateway might have to create that context. Essentially, the gateway decides to send StartTransaction at the moment the charger actually starts delivering current to a vehicle, and StopTransaction when it stops.
- Smart charging profiles from CSMS would map to setting a **max current parameter** on the charger. The mapping config should identify the Modbus register or ECHONET parameter that controls the charge current limit or power output. For example, maybe register 120 holds allowed amperage – the gateway would write to that when a `SetChargingProfile` is received (if the profile is a simple limit).

- For data values, the mapping can include scaling or unit conversion. E.g., ECHONET often provides power in Watts and cumulative energy in increments of 0.1 kWh (just an illustration); OCPP might expect Wh – the mapping logic would multiply or divide as needed. Another example: temperatures or voltages might need conversion from an integer register to a float, etc.

- **Mapping Types:** We need to handle two directions of mapping:

- **Device-to-OCPP (Incoming data/event):** E.g., charger signals something -> gateway sends OCPP message. This includes status changes, meter readings, and events like a fault. The mapping config should specify triggers. Possibly a simple approach is polling-based: the gateway periodically reads device values and compares to last known state – if changed, send an OCPP notification. Alternatively, if the protocol supports push (ECHONET notifications), the gateway can react to those asynchronously. Either way, mapping tells us what OCPP message to send when a certain condition is met. For instance, "if charging current > 0 and previously it was 0, send TransactionEvent (Started)" or "if fault code register is nonzero, send StatusNotification with Faulted and Report the error code."

- **OCPP-to-Device (Outgoing command):** E.g., CSMS sends command -> gateway writes to device. This covers RemoteStart, RemoteStop, change availability, etc. The mapping must identify the device action for each OCPP command. For example:

    - **RemoteStartTransaction**: No direct analog in device, so likely mapping will be: when CSMS requests a remote start for charger X, the gateway will pre-authorize internally and if the charger is idle, send whatever signal turns on power. If the charger needs a user present or physical action, this can be tricky – some chargers won't start just by command

unless a vehicle is connected. But assuming vehicle is connected and waiting, the gateway could close a relay or send a start command via Modbus.
    - **RemoteStopTransaction**: Mapping to turning off the output or stopping charging. E.g., set Modbus coil "ChargingEnable" to 0.
    - **ChangeAvailability**: (OCPP allows a station to be taken Inoperative or back InService). Mapping could be: if made Inoperative, disable the charger (maybe via a control register) or at least refuse any new sessions by the gateway. The gateway would also reflect this state in status (send StatusNotification "Unavailable"). If made operative, allow operations again.
    - **SetChargingProfile**: Map to setting current limit as described.
    - **Reset**: OCPP Reset command (soft or hard reset). The mapping might invoke a device reboot if possible. For example, some chargers might have a control to reboot or the gateway itself could just simulate a power cycle. If not available, the gateway might just report Reset executed (and perhaps reinitialize communications).

- **Mapping Storage and Management:** We plan to store mapping info in a structured way:

- Possibly have a **device profile template** for each type of device. e.g., "ChargerTypeA_mapping.yaml", "ChargerTypeB_mapping.yaml". Each profile defines how to handle that type's data. Then we have a list of actual devices in the system config, each referencing one of these profiles and providing instance-specific details (like its Modbus ID or IP address).
- This separation means adding support for a new charger model is as simple as creating a new mapping template and adding entries for the devices.
- The mapping configuration can be loaded on startup. We should also consider supporting **dynamic reload** (so an operator could update a mapping file and instruct the gateway to reload without full restart). This is helpful during commissioning when one might tweak mappings for correctness.

- For user-friendliness, a GUI or at least a documented schema for the mapping file is ideal. The user could use a web UI to input the registers/EPCs or choose from dropdowns if templates exist. (For initial implementation, this might be too much – so just a config file is fine, but designing with a future GUI in mind is good).

- **Commissioning Tools:** The gateway development team might create a simple tool to assist mapping configuration – for instance, a scanner that reads all relevant registers/ECHONET properties from a device and dumps values (to help identify what changes when the charger starts, etc.). This isn't part of the final product delivered to end users, but aids in creating the correct mapping for each new charger model encountered.

- **Real-World Example (Intesis Gateway):** The Intesis OCPP<->Modbus gateway uses a fixed mapping and a PC software called **Intesis MAPS** for configuration [27] [4]. Users can import templates and adjust parameters, then download the config to the gateway. This approach underscores the value of having pre-defined mappings for known device types and a user-friendly way to manage them. Our gateway should aim for similar flexibility. We may cite Intesis: "Templates can be imported and reused as often as needed, significantly reducing commissioning time." [3]. This indicates that having a library of device mappings and the ability to load them speeds up deployment.

- **Handling Differences in Device Capability:** The mapping layer may sometimes need to **emulate or ignore certain OCPP features** if the device doesn't support them. For example, if OCPP asks

for a **reservation to be made**, but the charger has no concept of reservation, the gateway could still accept the ReserveNow request and simply remember that reservation internally (and refuse a plug-in during the reserved period). If OCPP tries to set a display message and the charger has no screen, the gateway might return a status indicating not supported. The mapping config can flag such limitations. This way, the system remains OCPP-compliant (it responds to all messages properly) even if some are essentially no-ops at the device level.

- **Testing the Mapping:** It's vital to test each mapping with real or simulated devices. Unit tests could be written that feed in sample Modbus register sets or ECHONET property values and assert that the gateway would produce the correct OCPP JSON message, and vice versa for commands. This ensures the mapping logic is correct. Over time, a library of these test cases for each device type will ensure reliability when changes occur.

In summary, **the mapping design is all about flexibility and transparency**. By making it configuration-driven, we ensure that the gateway can adapt to various hardware without code changes. The structure might involve mapping tables, condition-action rules, and transformations, all defined externally. Initially, a well-documented config file approach is sufficient; as the product matures, a GUI or automated tool can assist with mapping generation.

Communication Methods: Evaluation and Recommendations

The gateway deals with multiple communication mediums: **TCP/IP (for both OCPP and Modbus TCP)**, **UDP (for ECHONET Lite)**, and **RS-485 serial (for Modbus RTU)**. Each has its pros and cons in terms of implementation complexity and maintainability. Below is a comparison and our proposed approach for each:

TCP/IP (e.g. OCPP over WebSocket, Modbus TCP)	- OCPP 2.0.1 uses WebSocket over TCP (with TLS). - Optional use for Modbus TCP to talk to chargers if they have Ethernet or via serial-to-TCP converters.	Advantages: **Reliable & Ordered:** TCP ensures all data arrives in order, simplifying protocol handling (no need for manual retransmissions or sequence tracking). **Mature Libraries:** We can leverage standard socket libraries; many high-level C++ libraries exist for WebSockets and TCP streams, reducing development effort. **Parallelism:** Each TCP connection is independent – can handle many in parallel (the OS handles the low-level multiplexing). This is ideal for our 100 OCPP connections or multiple Modbus TCP connections. **Ease of Debugging:** TCP traffic can be captured (e.g. with Wireshark) and is human-inspectable (especially for OCPP JSON over WebSocket). This aids maintenance and troubleshooting.	Challenges: **Overhead:** TCP has some overhead (connection establishment, etc.), but for persistent connections like ours, this is minor. WebSocket adds a bit of overhead, but OCPP 2.0.1 even supports compression to mitigate this. **Resource Usage:** Each connection consumes some memory/file descriptors. 100 connections is very manageable on modern hardware, but it's a consideration on a tiny embedded device. On our Linux gateway (likely reasonably powerful), this is fine. **Dependency on Network Stack:** Requires that IP networking is available and properly configured for all devices. In some industrial settings, configuring IP for each charger (or providing converters) is extra effort compared to a simple serial bus.	**Use TCP/IP wherever possible for ease and scalability.** - OCPP: **Use WebSocket over TCP** (with TLS) as mandated. This is straightforward with existing libraries and yields robust, persistent sessions [28] . - **Modbus:** If chargers support **Modbus TCP**, prefer it over RTU. It simplifies concurrency (each device is a separate socket) and can often run at faster data rates than serial. - If chargers only have RS-485, consider adding **serial-to-TCP gateways** for them. These devices convert RS-485 to an IP connection (effectively each charger appears at a distinct IP:port). This allows the gateway software to treat them like Modbus TCP, greatly simplifying logic (just open 100 sockets to 100 converters). This approach improves maintainability at the cost of extra hardware. If hardware addition is not desired or

Communication Interface	Usage in System	Pros (Maintainability & Ease)	Cons / Challenges	Recommendation
				too costly, then the gateway software will handle RS-485 directly (see below).

UDP/IP (ECHONET Lite communication)	- ECHONET Lite protocol runs over UDP (usually IPv4 multicast for discovery and unicast for control, on a standard port). - The gateway uses UDP to send requests to devices and receive their responses or event notifications.	Advantages: **Connectionless (Simple):** No need to maintain connection state per device – the gateway can use one socket to talk to many devices, just addressing packets to different IPs. This can simplify the code for multi-device handling, as the OS doesn't need to handle multiple connections. **Low Overhead:** UDP packets have minimal header, and no handshake. Good for simple query/response as in ECHONET. This also means slightly lower latency for single messages (no 3-way TCP handshake). **Broadcast/ Multicast:** UDP easily supports sending one packet to discover or instruct multiple devices. ECHONET leverages this for device discovery and group communications in some cases.	Challenges: **Unreliable:** UDP provides no guarantee of delivery, ordering, or duplicate protection. The gateway must implement any needed retries or acknowledgments at the application level. For ECHONET, the protocol itself defines responses for requests, but spontaneous notifications could be lost without detection. This requires careful design to ensure important updates (like StopTransaction signals) are not missed – possibly by polling as a backup. **Complexity in Parsing Multiple Responses:** If broadcasting a request (like discovery), many devices might respond at once. The gateway has to handle concurrent incoming UDP messages and match them to requests or devices. This is not too difficult (just requires non-blocking socket reads and a way to identify device by IP), but it's more manual than TCP's stream approach. **Network Considerations:** Multicast traffic might require enabling on	**Embrace UDP for ECHONET, with reliability measures.** - We will implement the ECHONET Lite protocol on UDP as required (there's no alternative transport in practice for ECHONET). This means writing a robust UDP handler: use non-blocking recv, handle timeouts, and implement **retry logic** for critical commands. For example, if we send a "Stop Charging" command via ECHONET and don't get a response in X milliseconds, resend it a couple of times before erroring out. - Keep the UDP handling code contained in the ECHONET adapter module. We will follow the ECHONET specification guidelines for request/response (which implicitly handles some reliability by design – each request expects a response) and for subscriber notifications.

Communication Interface	Usage in System	Pros (Maintainability & Ease)	Cons / Challenges	Recommendation
			network switches, and ensuring the Linux network stack is configured (e.g., joining the multicast group). If devices are on IPv6 (ECHONET supports IPv6 as well), additional configuration is needed. These can add to maintenance effort if not familiar.	 - From a maintenance perspective, ensure to log UDP communications (requests and responses) with timestamps. This helps diagnose any lost packet issues. Since UDP is stateless, our logs and perhaps a simple sequence number on requests (ECHONET has transaction IDs) will be key to troubleshooting. - Using UDP is not optional for ECHONET, but it's manageable. The development complexity is moderate (we need to craft and parse binary messages and handle timeouts). Given that we must support ECHONET, we accept UDP and mitigate its issues via careful coding.

RS-485 Serial (Modbus RTU via two-wire serial)	- Used for Modbus RTU communication with chargers that only have a serial port. RS-485 allows daisy-chaining multiple devices on one bus. - The gateway would use a serial interface (e.g., USB-to-RS485 adapter) to connect. Possibly multiple adapters for multiple buses.	Advantages: **Widely Supported by Legacy Devices:** Many industrial devices (including some chargers) only have Modbus RTU. Supporting it is often required to integrate older hardware without extra converters. **No IP Setup Needed:** Devices are identified by station IDs, not IP addresses, which can simplify network setup. For on-site installations with short distances, a single cable bus can be simpler than setting up an Ethernet switch network, potentially reducing hardware cost (one cable vs multiple Ethernet drops or converters). **Multiple devices, one port:** Up to 32 devices (or more with repeaters) can share one RS-485 line. In theory, this means only one serial port is needed for many devices (though performance will suffer if too many). This centralized wiring can be an advantage in some setups.	Challenges: **Programming Complexity:** Handling serial timing and bus arbitration is more complex than TCP. The gateway must ensure only one master (itself) drives the bus at a time and must wait for each response or timeout before moving on. Tuning inter-frame delays and turnaround times is critical – too fast and you overrun devices, too slow and you waste time. **Speed & Throughput:** Serial links typically run at 9.6 to 115 kbps. Even at 115k, if you have tens of devices, the shared bandwidth and sequential polling make it much slower than Ethernet (100 Mbps+). This means meter reads and status updates cannot be as frequent for 100 devices on one line. It may be acceptable if each device is polled say once per 5-10 seconds, but it's a limitation. For 100 devices, one might need to segment into multiple buses to get acceptable performance. **Reliability and Debugging:** Noise or wiring issues can cause communication errors. Diagnosing RS-485 issues is	**Support RS-485 Modbus RTU, but minimize its use if possible.** - If some chargers only have RS-485, we will implement Modbus RTU communication for them. Use a stable library or driver for Modbus RTU to avoid low-level pitfalls. The code will handle retries on checksum errors or timeouts, and mark devices offline if unresponsive. - **Segment the network:** Do not put all 100 devices on one serial line. It's better to use a few RS-485 adapters (the Linux device could have multiple serial ports via USB). E.g., 4 ports with ~25 devices each. This can quadruple the throughput by polling in parallel on each port (each port handled by a separate thread or asynchronous handler). It also limits the impact of a fault – e.g., a short or noise on one segment doesn't bring down all devices. - **Consider RS-485 vs.**

		trickier – one might need an oscilloscope or special serial logging to see collisions or signal integrity problems. This is less straightforward than capturing TCP packets. Also, adding or removing a device from the bus requires careful attention to bus biasing and termination. These factors make maintenance in the field slightly more demanding. **Scaling Code-wise:** If we keep 100 devices on one thread polling sequentially, that could become a bottleneck. We might need to implement a non-blocking I/O or a state machine to concurrently handle multiple requests even on one bus (though Modbus RTU prevents true concurrency, we could interleave queries to different buses). This adds complexity to the code design.	**Converters:** If maintainability is a major concern, suggest using **Modbus TCP gateways** for the RS-485 devices (as noted under TCP/IP). Many off-the-shelf small devices can convert Modbus RTU to TCP (some chargers might even come with an option for an Ethernet module). This shifts the burden to a well-tested converter and lets our gateway treat everything as TCP. The trade-off is cost per charger and additional points of failure (the converter itself). If the project can afford it, this is ideal for maintainability. If not, we proceed with native RS-485 support. - In implementing RS-485, ensure thorough **testing on the real bus** with multiple devices to tune timing. Also implement logging of request/response per device with timestamps, which can help in diagnosing if one device is slowing down

Communication Interface	Usage in System	Pros (Maintainability & Ease)	Cons / Challenges	Recommendation
				communication (e.g., taking too long to respond) or if timeouts are happening. - **Maintenance tip:** Document the bus layout and device IDs clearly, so that field technicians know which device corresponds to which ID on which port. This aids in troubleshooting wiring issues or device replacements.

To summarize the communication policy: - **OCPP side:** Use **TCP/IP (WebSocket over TLS)** – the standard method. This is non-negotiable since OCPP 2.0.1 specifies WebSocket as the transport (and SOAP optional, but we will use WebSocket/JSON which is most common). We will utilize the persistent, stateful connection benefits of WebSockets to handle bi-directional messaging efficiently [29] . - **Device side:** Use **Modbus TCP/IP** wherever available for easier scalability. For ECHONET, UDP is the only way, so we implement it with care. For Modbus RTU, we support it but aim to limit complexity by possibly introducing additional hardware (converters) if acceptable, and by using multiple lines to ease the load.

Maintainability Consideration: Each communication method should come with proper abstraction. The gateway's core logic (mapping layer and OCPP handling) should not need to worry if a charger is on TCP or serial. We will implement a uniform interface for "ChargerConnection" that hides whether it's via Modbus TCP, Modbus RTU, or ECHONET. This interface can provide read/write operations (with timeouts) and the higher layers just call those. Internally, different strategies (threads, async) are used, but to the mapping and OCPP logic, it's uniform. This abstraction improves maintainability – e.g., if one day a new communication channel is needed (say CAN bus or some other protocol), we can add a new adapter class without rewriting the core logic.

In conclusion, the communication approach is to **leverage networked (IP-based) communication for reliability and simplicity**, and use traditional serial communication only as necessary, with mitigation strategies for its complexities. By doing so, we aim for a robust system that is easier to develop, debug, and maintain in the long run, while still meeting the requirement of supporting legacy interfaces.

Conclusion

Bringing together the above elements, we propose a **C++ OCPP 2.0.1 Gateway Middleware** with the following key characteristics:

- **Robust Core OCPP Implementation:** A focus on **Core profile** features and most-used OCPP functionalities ensures the gateway will cover fundamental operations needed for EV charging management from day one [12] [2] . Additional features are planned in phases, aligned with their real-world utility (smart charging, etc.) to incrementally enrich the system.

- **Scalable, Modular Architecture:** A multi-threaded or asynchronous architecture that can manage ~100 concurrent device communications and OCPP sessions. The architecture is modular – separating OCPP protocol handling, device-specific logic, and the translation layer – which improves clarity and maintainability. The provided architecture diagram illustrates these modules and how they interact in the system context.

- **Flexible Mapping Engine:** A configurable mapping system that will enable integration of various charger models (ECHONET Lite or Modbus) without code changes. By externalizing the mapping (with templates and config files), the solution becomes adaptable. This approach is in line with industry solutions that use reusable templates to speed up commissioning [3] . It will allow the gateway to be deployed across different installations with minimal customization effort – simply by adjusting configuration rather than modifying software.

- **Communication Strategy for Reliability:** Utilizing proven communication methods (TCP/IP and WebSockets) for network interactions and carefully managing lower-level protocols (UDP, RS-485) where needed. The recommendations provided ensure that the gateway optimizes for easier implementation and maintenance: using TCP for Modbus whenever possible, and structuring RS-485 support in a way that isolates complexity. We cite that Modbus can run over many media [30] – and we choose the medium that best balances ease and necessity in each case (IP for ease, serial if it's the only option).

- **Linux-Based Deployment:** The software will be developed and tested on Linux, taking advantage of the OS's networking and multi-tasking capabilities. It can run as a service (e.g., systemd daemon) for auto-start and supervision. Linux provides stability and the necessary drivers (for serial ports, TLS libraries, etc.) out of the box, simplifying the platform requirements.

- **Security and Maintainability:** All OCPP communication will be secured via TLS. Internally, the design will include logging, error handling, and possibly self-recovery mechanisms (e.g., auto-reconnect to CSMS or devices) to ensure resilience. For maintainers, clear logs and possibly a monitoring dashboard will make it easier to support the system. The gateway should also expose some health metrics (maybe via an HTTP endpoint or simply logs) so that operators know how many devices are connected, if any errors are occurring, etc.

Finally, by providing a requirements list, architecture diagram, feature matrix, mapping approach, and comms comparison, we have a comprehensive blueprint. This blueprint can serve as a starting point for the detailed design and implementation phase. All critical design decisions were made keeping in mind the **frequency of use in real-world scenarios** and the **ease of development/maintenance**, ensuring that the gateway will be both useful and practical. With this foundation, the development team can proceed to implementation confident that the key considerations and industry best practices are accounted for.

Sources:

- Open Charge Alliance – OCPP 2.0.1 Feature Profiles and Core Requirements [1] [11]
- Intesis (HMS) – OCPP to Modbus Gateway features (illustrating minimal OCPP operations supported) [2]
- Intesis – Configuration via templates for OCPP/Modbus gateway (Intesis MAPS tool) [3] [4]
- ResearchGate – Modbus over various media (serial, TCP/IP) flexibility [30]
- Medium (Siddharth Sabron) – Scalable OCPP Server design (event-driven WebSocket benefits) [28]

---

[1] [6] [10] [11] [12] [15] [16] [18] [19] [20] [21] [23] [24] OCPP for Resource-Constrained Devices
https://openchargealliance.org/wp-content/uploads/2025/05/ocpp_2_x_minimal_footprint-v14.pdf

[2] [4] OCPP to Modbus TCP & RTU Server Gateway - 01 device by INTESIS - Ivory Egg
https://ivoryegg.com.au/shop/products/inmbsocp0010100-ocpp-to-modbus-tcp-rtu-server-gateway-01-device

[3] [9] [27] OCPP to Modbus TCP & RTU Server Gateway
https://www.hms-networks.com/p/inmbsocp0010100-ocpp-to-modbus-tcp-rtu-server-gateway

[5] [28] [29] Designing a Scalable Open Charge Point Protocol server | by Siddharth Sabron | Medium
https://medium.com/@siddharth.sabron/designing-a-scallable-open-charge-point-protocol-server-63694204efbe

[7] [8] [14] [17] [25] [26] What is new in OCPP 2.0.1
https://openchargealliance.org/wp-content/uploads/2024/01/new_in_ocpp_201-v10.pdf

[13] OCPP 1.6 vs 2.0 vs. 2.1 Comparing: Benefits, Limitations, and ...
https://lembergsolutions.com/blog/ocpp-16-vs-20-vs-21-comparing-benefits-limitations-and-adoption-trends

[22] EV Connect Announces OCPP 2.0.1 Certification
https://www.evconnect.com/news/ocpp

[30] Modbus architecture. | Download Scientific Diagram
https://www.researchgate.net/figure/Modbus-architecture_fig1_357232645