Implementing a Server-Sent Events (SSE) Pub/Sub Pattern with Python

Wolfgang Spahn

Abstract

In this tutorial, we walk through how we implemented the pub/sub (publish/subscribe) pattern using Server-Sent Events (SSE) in Python. The solution provided creates a thread-safe mechanism for managing multiple clients via SSE, using Python's `queue.Queue` for message handling and threading for concurrent operations.

Tutorial

Part 1

Implementing a Server-Sent Events (SSE) Pub/Sub Pattern with Python

Overview of the Solution We implemented a `MessageAnnouncer` class that: 1. **Publishes messages** to clients through the `announce()` method. 2. **Allows clients to subscribe** via the `listen()` method, where each client receives messages in real-time. 3. **Manages threads** safely and pings clients periodically to keep the connection alive.

Additionally, we will see how the implementation uses elements like thread locks to avoid concurrency issues and ensure robustness behind a reverse proxy like Nginx.

Key Components of the Code

1. `format_sse` function This function formats messages according to the SSE protocol. It takes two parameters: the data to send and an optional event name. The SSE protocol sends data in a specific format that requires each piece of data to be prefixed with `data:` and an event name, if provided, with `event:`.

```python
def format_sse(data: str, event=None) -> str:
    """Formats a string and an event name in order to follow the
       event stream convention."""
    msg = f'data: {data}\n\n'
    if event is not None:
        msg = f'event: {event}\n{msg}'
    return msg
```

Example output for sending data:

```python
format_sse(data=json.dumps({'abc': 123}), event='Jackson 5')
# Returns: 'event: Jackson 5\ndata: {"abc": 123}\n\n'
```

2. `MessageAnnouncer` class This class manages the pub/sub mechanism and handles clients subscribing to and receiving updates.

```python
def __init__(self):
    logger.debug("SSE -- init MessageAnnouncer while starting\
```

```python
                    ping thread.")
    self.listener_locks = {}
    self.start_ping()
```

Initialization (`__init__`)

- `self.listener_locks`: A dictionary that maps each client's message queue to a thread lock, ensuring thread-safe access to the queue.
- The `start_ping()` method starts a thread that periodically sends ping messages to all connected clients to keep the connection alive.

`listen()` method This method is called when a client subscribes to the message stream. It creates a `queue.Queue` for each client that will block until a new message is available.

```python
def listen(self):
    """Returns a queue.Queue that blocks until a new item is
       added to the SSE stream."""
    q = queue.Queue(maxsize=5)
    self.listener_locks[q] = threading.Lock()
    q.put_nowait(format_sse(
        data="SSE has successfully connected.", event="START"))
    return q
```

- A new `queue.Queue` with a maximum size of 5 is created for each client. This queue blocks until new messages are available.
- The client is immediately sent a connection confirmation message (`SSE has successfully connected.`).

`announce()` method The `announce()` method publishes new messages to all clients that are currently connected.

```python
def announce(self, msg):
    """Announces a new item to the SSE stream."""
    to_remove = []
    for q, lock in list(self.listener_locks.items()):
        with lock:
            try:
                q.put_nowait(msg)
            except queue.Full:
                to_remove.append(q)
            except Exception as e:
                logger.error(f"Error sending message: {e}")
    for q in to_remove:
        del self.listener_locks[q]
```

- It iterates through each client's message queue and attempts to push the new message.
- If the queue is full, the client is marked for removal to prevent further issues.
- After sending the message, disconnected or full queues are removed from the system.

`broadcast()` method This method is similar to `announce()` but ensures that messages are formatted according to SSE conventions.

```python
def broadcast(self, message):
    """Sends a message to all connected clients."""
    to_remove = []
    for q, lock in self.listener_locks.items():
        with lock:
            try:
```

```
                q.put_nowait(format_sse(data=message))
            except queue.Full:
                to_remove.append(q)
            except Exception as e:
                logger.error(f"Error sending message: {e}")
                to_remove.append(q)
    with threading.Lock():
        for q in to_remove:
            del self.listener_locks[q]
```

- It formats the message using the `format_sse()` function before sending it to all clients.

`start_ping()` and `ping_clients()` This starts a background thread to send periodic "ping" messages to clients. This helps maintain connections behind reverse proxies like Nginx, which may close idle connections without some form of activity.

```
def start_ping(self):
    threading.Thread(
        target=self.ping_clients, daemon=True).start()

def ping_clients(self):
    while True:
        to_remove = []
        for q, lock in self.listener_locks.items():
            with lock:
                try:
                    q.put_nowait(
                        format_sse(data="ping", event="PING"))
                except Exception as e:
                    to_remove.append(q)
        with threading.Lock():
            for q in to_remove:
                try:
                    del self.listener_locks[q]
            time.sleep(1)
```

- The `ping_clients()` method sends a `ping` event every second to each client. If a client is disconnected or if there are any issues sending the ping, the client is removed from the system.

Handling Robustness Behind Nginx

For this SSE implementation to work well behind Nginx, it is crucial to keep the connection alive and ensure that clients do not hang due to long idle periods. We ensure this by:

1. **Ping Mechanism**: The `ping_clients()` method sends periodic pings (`"event: PING"`) to maintain the connection, preventing timeouts that might occur due to inactivity.
2. **Thread Safety**: We use locks (`threading.Lock()`) for each listener's queue to ensure that multiple threads do not access the same resources simultaneously, avoiding race conditions and deadlocks.
3. **Queue Management**: Each client has its own `queue.Queue`, which prevents blocking between clients. When a queue becomes full (client stops consuming messages), it is safely removed from the system, ensuring no memory leaks.

Conclusion

This implementation of the pub/sub pattern using SSE in Python is robust, thread-safe, and capable of handling multiple clients concurrently. The use of queues for each client and thread locks ensures that the system remains responsive and reliable. Additionally, the ping mechanism ensures connections are kept alive behind reverse proxies like Nginx.

This setup is well-suited for real-time applications where updates need to be pushed to many clients, such as live notifications or updates in web applications.

Part 2

Implementing an SSE Manager with Multiprocessing in Python

In this tutorial, we will walk through the implementation of a Server-Sent Events (SSE) Manager using Python's `multiprocessing` library. This SSE Manager is designed to support multiple processes accessing a shared resource for sending and receiving SSE messages. It is based on Python's `multiprocessing.managers.BaseManager`, which allows processes to interact with shared objects.

This approach enables safe communication between processes by ensuring that the operations on the SSE message announcer are thread-safe. We'll also discuss the role of each component in this code.

Key Concepts

- **Multiprocessing Manager (`BaseManager`)**: Allows different processes to interact with shared resources like queues and locks.
- **Thread-Safety**: Achieved by using a `Lock` object to synchronize access to shared resources.
- **SSE Server**: The server listens for new connections, sends messages to clients, and manages the number of listeners.
- **Logging**: Tracks the flow of the application for debugging and monitoring.

Let's dive into the individual components of the code.

1. Setting Up the `SSEManager`

The `SSEManager` is derived from Python's `BaseManager`, which allows remote processes to call methods on the manager and interact with its resources.

```
class SSEManager(BaseManager):
    pass
```

The `SSEManager` does not need additional methods of its own, but it will allow the registration of functions such as `sse_listen`, `sse_put`, and `get_listener_count` so that they can be accessed remotely by other processes.

2. The `start_sse` Function

This function starts the SSE server and initializes the required resources. It sets up logging, creates the `MessageAnnouncer` instance, and registers the necessary methods for SSE communication.

Logging Setup The logging setup allows tracking of events, making it easier to debug and monitor the application's behavior.

```
setup_logging()
logger.info("SSE -- process: start")
```

Creating a Lock We create a `Lock()` to ensure thread-safe access to shared resources. This is crucial because the SSE server is designed to handle multiple clients in a concurrent environment.

```
lock = Lock()  # Create a mutex lock to ensure thread-safe operations.
```

Creating the `MessageAnnouncer` The `MessageAnnouncer` object is responsible for broadcasting messages to clients and keeping the connection alive using a ping mechanism (as seen in the previous code).

```
sse = MessageAnnouncer()
```

SSE-Related Functions

1. `sse_listen()`: This method listens for new SSE messages. It returns a blocking queue from the `MessageAnnouncer` that holds the next message for the client. The use of the `lock` ensures thread-safety when multiple clients try to subscribe.

```python
def sse_listen():
    with lock:
        logger.debug("SSE -- process: listen")
        message = sse.listen()
        logger.debug(f"SSE -- received: {message}")
        return message
```

2. `sse_put()`: This function allows the server to broadcast a new item to the SSE stream, i.e., send a message to all connected clients. The message is announced using the `announce()` method of the `MessageAnnouncer`.

```python
def sse_put(item):
    with lock:
        logger.debug(f"SSE -- Sending SSE message: {item}")
        sse.announce(item)
```

3. `get_listener_count()`: This method returns the current number of listeners connected to the SSE server. It accesses the `listener_locks` dictionary inside the `MessageAnnouncer` to get the count of active listeners.

```python
def get_listener_count():
    with lock:
        listener_count = len(sse.listener_locks)
        logger.debug(
            f"SSE -- Current listener count: {listener_count}")
        return listener_count
```

Registering Methods with `SSEManager` Each of the functions (`sse_listen`, `sse_put`, `get_listener_count`) must be registered with the `SSEManager` so that they can be accessed remotely by other processes.

```python
SSEManager.register("sse_listen", sse_listen)
SSEManager.register("sse_put", sse_put)
SSEManager.register("get_listener_count", get_listener_count)
```

Starting the SSE Manager We create an instance of the `SSEManager`, specifying the address (`127.0.0.1`) and the port where the manager will be listening for connections. The `authkey` provides a layer of security to authenticate connections to the server.

```python
manager = SSEManager(address=("127.0.0.1", sse_port),
                     authkey=b'sse')
logger.info(
    f"SSE -- serving SSE server at address {manager.address}")
```

Once the server is ready, it calls `serve_forever()` to start listening for incoming requests.

```
ready_event.set()
server = manager.get_server()
server.serve_forever()
```

3. Handling Errors

The `start_sse` function is wrapped in a try-except block to handle any errors that may arise during the initialization or execution of the SSE manager.

```
except Exception as e:
    logging.error(
        f"SSE Manager -- Failed to start SSE server: {e}")
```

4. Summary of the Registered Methods

At the bottom of the code, we also register the SSE functions globally, allowing them to be accessed from another process when necessary.

```
SSEManager.register("sse_listen")
SSEManager.register("sse_put")
SSEManager.register("get_listener_count")
```

This allows processes outside of the main one to remotely invoke these methods, ensuring that the SSE server can communicate with clients or other parts of the application running in parallel.

Conclusion

This implementation of an SSE Manager using Python's multiprocessing capabilities provides a robust and thread-safe way to manage real-time client connections via Server-Sent Events. By leveraging `BaseManager` and `Lock`, the solution ensures that concurrent access to shared resources like queues and listener counts is properly synchronized.

The setup of the SSE manager allows it to be run as a separate process, enabling inter-process communication and scalability. This makes the system suitable for applications that require real-time updates, such as live data feeds or notification systems.

Key Takeaways:

- **Multiprocessing with `BaseManager`**: Ensures safe communication and sharing of objects between processes.
- **Thread-safety with `Lock()`**: Synchronizes access to shared resources like queues to avoid race conditions.
- **Logging**: Helps in monitoring and debugging the system.
- **Robust SSE Management**: Ensures real-time communication with multiple clients, keeping connections alive with periodic pings.

This architecture makes it easy to scale SSE servers and handle multiple clients simultaneously in a safe and efficient way.

Part 3

Integrating Server-Sent Events (SSE) with Flask and Multiprocessing

This tutorial covers how to integrate Server-Sent Events (SSE) with a Flask application using multiprocessing and inter-process communication. This approach involves using `SSEManager`, which manages SSE message distribution across processes, and Flask to serve the messages to connected clients.

Key points: - **SSEManager** is used to handle SSE message broadcasting in a separate process. - **Flask** is used to stream these events to the clients over HTTP. - **Inter-process communication** via `multiprocessing.managers.BaseManager` allows interaction between Flask and the SSE server running in another process.

1. Key Concepts

- **SSE**: Server-Sent Events are a way to push updates from the server to the client over a long-lived HTTP connection.
- **Flask**: A lightweight web framework used to serve the SSE stream to clients.
- **`SSEManager`**: A manager that handles SSE messages in a separate process. It allows interaction with the SSE announcer from other processes through registered methods.

2. SSE Manager Interaction

This part of the code connects to the remote SSE server (which could be on the same machine but in a different process) and allows Flask to forward messages to subscribers or receive messages for broadcasting.

The `notify_subscribers` function This function is used to notify all subscribers by sending a message to the SSE server. It formats the message according to the SSE protocol and forwards it to the `sse_put()` method in `SSEManager`.

```python
def notify_subscribers(sse_manager, data, event_type=None):
    # Connect to the remote SSE server
    sse_manager.connect()  # Connect to the SSE server
    # Format the data to conform to the SSE format
    msg = format_sse(data=json.dumps(data), event=event_type)
    # Send the message to the SSE server (put it in the queue)
    sse_manager.sse_put(msg)  # Broadcast to all clients
```

- **`sse_manager.connect()`**: Establishes the connection to the SSE server, which is running in a different process.
- **`format_sse()`**: Converts the `data` into the SSE format, including optional event types.
- **`sse_manager.sse_put()`**: Sends the message to all connected listeners via the SSE server.

The `stream` function This function is responsible for streaming SSE messages to the connected clients. It listens for new messages from the SSE server and forwards them to the client over an HTTP connection.

```python
def stream(sse_manager):
    """Stream Server-Sent Events (SSE) to the client."""
    sse_manager.connect()  # Connect to the SSE server
    # Get the message queue from the SSE server
    messages = sse_manager.sse_listen()

    try:
        while True:
            # Wait for a new message from the queue
            # (blocks until a message is received)
            msg = messages.get()
            if msg is None:
                logger.error(
                    f"stream received None message: {msg}")
                break
            # Parse the received SSE message
            msgDict = parse_sse_msg(msg)
            if msgDict is None:
                logger.error(
                    f"stream received invalid SSE message: {msg}")
                break
```

```python
            # Yield the message to the client
            if 'data' in msgDict:
                yield f"data: {msgDict['data']}\n\n"
            elif 'data' in msgDict and 'event' in msgDict:
                yield f"event: {msgDict['event']}\ndata: {msgDict['data']}\n\n"
            else:
                logger.error(
                    f"stream received non-conformant SSE message: {msgDict}")
                yield f"error: message\n{msg}\n\n"
    except Exception as e:
        logger.error(f"Error during SSE communication: {e}")
        return Response("Error", status=500)
```

- **`sse_manager.sse_listen()`**: Connects to the SSE server and listens for new messages. This returns a blocking queue that waits for new messages.
- **`messages.get()`**: Retrieves a new message from the queue (blocks until a message is available).
- **`yield`**: Sends the message to the client via a long-lived HTTP connection.
- **Error Handling**: If an invalid message is received or an exception occurs, the function logs an error and returns a 500 response.

The `parse_sse_msg` function This helper function parses the received SSE message into a dictionary. It expects the message to follow the SSE format and extracts the key-value pairs.

```python
def parse_sse_msg(msg):
    try:
        lines = msg.strip('\n').split('\n')
        keyVals = [li.split(":") for li in lines]
        keyVals = [(kv[0], kv[1]) for kv in keyVals]
        return dict(keyVals)
    except (IndexError, ValueError) as e:
        logger.error(f"Invalid SSE message: {e}")
        return None
```

- **Message Format**: The message is expected to have the format `key: value` on each line. This function splits the message and constructs a dictionary of key-value pairs.
- **Error Handling**: If the message format is invalid, it logs the error and returns `None`.

3. Setting Up SSE with Flask

The `setup_sse_listen` function integrates the SSE manager with Flask, allowing the Flask app to interact with the SSE server.

```python
def setup_sse_listen(app, sse_port):
    sse_manager = SSEManager(address=("127.0.0.1", sse_port), authkey=b'sse')
    logger.info(f"remote sse manager is serving at: {sse_manager.address}")
    # Ensure app has an 'extensions' attribute
    if not hasattr(app, "extensions"):
        app.extensions = {}
    # Register the SSE manager with the app
    app.extensions["sse-manager"] = sse_manager
    return sse_manager
```

- **SSE Manager**: Initializes an `SSEManager` instance, specifying the address and port for communication. The `authkey` ensures security when connecting to the server.
- **Flask Integration**: The SSE manager is registered as an extension of the Flask app, making it available globally within the application.

- `sse_port`: The port on which the SSE server is running (could be a different process but on the same machine).

4. Handling the Remote/Local Processes

In this setup, **"remote"** refers to different processes on the **same machine**. The SSE manager and the Flask app can run in separate processes but share resources through inter-process communication. This is achieved using the `multiprocessing.managers.BaseManager`, which allows methods like `sse_listen()` and `sse_put()` to be called from different processes.

- **`SSEManager.connect()`**: Establishes a connection between the Flask process and the SSE manager (which is running in another process).
- **`sse_put()` and `sse_listen()`**: These methods are registered in the SSE manager and allow Flask to communicate with the SSE server to send and receive messages.

5. Streaming SSE to Clients

The Flask route for SSE would look like this:

```python
@app.route("/events")
def sse_stream():
    sse_manager = app.extensions["sse-manager"]
    return Response(stream(sse_manager),
                    content_type="text/event-stream")
```

- **`/events`**: This route streams SSE messages to the client.
- **`Response`**: The `stream()` function continuously yields messages to the client, and Flask keeps the connection open with the `text/event-stream` content type.

Conclusion

This setup allows Flask to handle Server-Sent Events in a multiprocessing environment. The SSE manager runs in a separate process to handle message broadcasting, while Flask streams these messages to clients over HTTP.

Key Takeaways: - **Multiprocessing**: The SSE manager runs in a separate process, but Flask can communicate with it through inter-process communication. - **Flask SSE**: Flask handles the HTTP connection and streams messages to clients. - **Thread-Safe Access**: The `Lock` and `BaseManager` ensure that resources are accessed safely across processes.

This architecture provides a scalable way to implement real-time updates using SSE, especially in applications that require constant communication with multiple clients.

Final Remark

How the Three Modules Interact

In this system, there are three primary components: **`manager`**, **`announcer`**, and **`routes`** (better referred to as the **Flask handler**). Each module plays a distinct role in handling Server-Sent Events (SSE) within a Flask application using multiprocessing. Here's how they interact:

1. **`announcer.py` (Message Broadcasting Logic)**:
    - This module defines the **`MessageAnnouncer`**, which is responsible for managing SSE streams and broadcasting messages to clients.
    - It provides two critical methods:
        - **`listen()`**: This method creates a message queue for each client that blocks until a new message is added.
        - **`announce()`**: This method broadcasts messages to all connected clients by placing them in the respective queues.

- The `MessageAnnouncer` also includes a ping mechanism to keep connections alive, ensuring robust communication with clients, even behind reverse proxies like Nginx.

2. **`manager.py` (SSE Management and Inter-process Communication)**:
   - **`SSEManager`** is implemented using Python's `multiprocessing.managers.BaseManager`. This allows for inter-process communication by exposing the methods of the **`MessageAnnouncer`** to other processes.
   - The **`start_sse`** function initializes the SSE server in a separate process. It registers the methods `sse_listen()`, `sse_put()`, and `get_listener_count()` so that they can be called remotely by other processes, including Flask routes.
   - By running in a separate process, **`SSEManager`** handles multiple clients and manages the distribution of messages (via `sse_put()` and `sse_listen()`) concurrently and safely across processes using locks.
3. **`routes.py` (Flask Handlers for SSE)**:
   - This module integrates the SSE functionality into a Flask application and handles client-side HTTP connections.
   - The **`notify_subscribers()`** function interacts with the `SSEManager`, forwarding messages to the SSE server, which in turn broadcasts them to all connected clients.
   - The **`stream()`** function listens for new messages from the SSE server and streams them over HTTP using the SSE protocol. It keeps the connection alive and sends updates to the client as they are received.
   - **`setup_sse_listen()`** ties the Flask app to the `SSEManager`, allowing Flask routes to access the SSE server running in another process.

Interaction Flow

- **Client Subscription**:
  1. When a client connects to the Flask route for SSE (e.g., `/events`), the **`stream()`** function is called. It requests the `sse_listen()` method from the **`SSEManager`** to get a message queue from the **`MessageAnnouncer`**.
  2. The **`MessageAnnouncer`** (from `announcer.py`) adds the client to its list of listeners and starts sending messages to this queue.
- **Message Broadcasting**:
  1. When new data needs to be sent to clients, **`notify_subscribers()`** is called, which interacts with the **`SSEManager`** to call the `sse_put()` method.
  2. The **`SSEManager`** calls `announce()` in the **`MessageAnnouncer`**, which then broadcasts the message to all connected clients via their individual queues.
- **Client Message Reception**:
  1. The **`stream()`** function continuously listens for new messages in the client's queue (provided by the **`MessageAnnouncer`**).
  2. As soon as a new message is placed in the queue, the message is streamed to the client's HTTP connection in the SSE format.

Summary of Module Responsibilities:

- **`announcer.py`**: Core logic for managing clients, broadcasting messages, and keeping connections alive.
- **`manager.py`**: Provides the inter-process communication layer, allowing Flask to communicate with the SSE server running in a separate process via `SSEManager`.
- **`routes.py`**: Flask routes for handling SSE client connections and forwarding messages to the SSE server for broadcasting.

Together, these three modules form a cohesive system where **`announcer.py`** manages the actual SSE logic, **`manager.py`** acts as the bridge for inter-process communication, and **`routes.py`** handles client requests and streaming through Flask.