ChatGPT

Worker Fails to Register with Orchestrator (RPCError: `CHANNEL_CLOSED`)

Problem Description

When starting the **worker** service, it fails to register itself with the **orchestrator**. The worker's log shows an error:

```
Worker failed to register with orchestrator: RPCError: CHANNEL_CLOSED: channel
closed
    at ProtomuxRPC._onclose ...
```

It also prints the worker's RPC public key (a hex string). This indicates that the worker's attempt to connect to the orchestrator over Hyperswarm RPC was unsuccessful – the RPC channel was closed almost immediately. As a result, the orchestrator doesn't list any registered workers, and inference requests cannot be dispatched.

Cause Analysis

This issue is usually caused by a **misconfiguration of the orchestrator's public key in the environment or a startup timing issue**:

- **Orchestrator Public Key Mismatch:** The worker uses Hyperswarm RPC to connect to the orchestrator. It needs the orchestrator's **public key** to do so. In the code, the worker reads the `ORCHESTRATOR_PUBLIC_KEY` from the environment and tries to connect using it [1]. If this key is missing or incorrect, the worker will be unable to find the orchestrator on the DHT network, causing the RPC connection to fail (yielding the `CHANNEL_CLOSED` error) [2].

Why this happens: By default, the orchestrator generates a new random key pair on each run unless you configure it to use a deterministic key. The Docker Compose file is set up with placeholders to provide a static key via environment variables [3]. If you don't set these in a `.env` file (or environment), the orchestrator will use a random key each time. In that case, the worker's `ORCHESTRATOR_PUBLIC_KEY` (if left blank or set to an old value) will not match the actual key of the running orchestrator. This mismatch means the worker is "dialing" the wrong address on the Hyperswarm DHT. The connection attempt then fails, as no orchestrator is listening at that key, resulting in the `RPCError: CHANNEL_CLOSED`.

Note that if only an `ORCHESTRATOR_PUBLIC_KEY` was provided without the corresponding secret key, the orchestrator code **ignores it** and still generates a new key pair. The code explicitly requires both `ORCHESTRATOR_PUBLIC_KEY` and `ORCHESTRATOR_SECRET_KEY` to use a fixed keyPair; otherwise it falls back to using `ORCHESTRATOR_SEED` or a random key [4]. This means setting just the public key in the env

(and not the secret) is not sufficient – the orchestrator will not actually use that provided public key, leading to a desynchronization between what the worker thinks the key is and what the orchestrator actually uses. In your case, the orchestrator log showed a public key (`f504d8...`) that likely did **not** match what the worker was using. The worker's own log printed a different key (`f4fac8...`) as its connection target or its identity, confirming a mismatch.

- **Startup Ordering (Race Condition):** Another contributing factor can be the order in which the services start. In the current Compose setup, the **worker** does not explicitly wait for the orchestrator to be ready before attempting registration. The worker container might come up and immediately try to register at a time when the orchestrator's RPC server isn't fully listening yet. The code does the registration only once on startup (it doesn't retry on failure) [5]. If the orchestrator isn't accepting connections at that exact moment, the worker's `client.request('register-worker', ...)` call will fail. This can produce a similar error (`CHANNEL_CLOSED`) because no channel could be established. In the Compose file, the orchestrator has a healthcheck and other services (like gateway, http-bridge) wait for it, but the worker service currently only depends on Redis and not on the orchestrator [6]. This means the worker can start in parallel with (or even before) the orchestrator. If orchestrator startup is slightly delayed (e.g., database migrations or just process initialization), the worker's registration attempt will hit a closed channel.

In summary, the **primary issue** is that the worker isn't connecting with the correct orchestrator public key (or at the correct time). The `CHANNEL_CLOSED` error is essentially saying "the RPC connection could not be established or was immediately closed," consistent with either a wrong key (no such peer) or the peer not being ready.

Solution

To fix the worker registration failure, you should address the configuration and startup order:

1. Configure a Static Orchestrator Key Pair:
Ensure that all services (orchestrator, worker, gateway, etc.) agree on the orchestrator's public key. The framework expects you to supply a deterministic key so that the orchestrator's identity is known in advance to the other components. There are a couple of ways to do this:

- **Provide Orchestrator Public/Secret Keys via `.env`:** Generate a Noise keypair (32-byte public and 32-byte secret in hex) and set them in your environment. In the `docker-compose.yml`, the orchestrator service has environment variables for `ORCHESTRATOR_PUBLIC_KEY` and `ORCHESTRATOR_SECRET_KEY` (as well as an optional seed) [3]. You should fill these in (e.g., in a `.env` file at the project root, as the Compose config will pick that up [7]). For example:

```
ORCHESTRATOR_PUBLIC_KEY=<your-orchestrator-public-key-hex>
ORCHESTRATOR_SECRET_KEY=<your-orchestrator-secret-key-hex>
```

Make sure the keys are a valid pair. Once these are set, the orchestrator will use this deterministic keypair on startup (instead of generating a new one). The worker and other services will receive the `ORCHESTRATOR_PUBLIC_KEY` from the same environment, so they will connect to the correct, fixed key. You should see the orchestrator log the same public key on each run, and the worker's registration call

should succeed. (If you need to generate a keypair: one way is to use a cryptographic library or Node.js script. For instance, using Node's crypto module or libsodium to create an X25519/Curve25519 key pair. Ensure you provide the correct 64 hex characters for each of public and secret keys.)

- **Use a Deterministic Seed:** Alternatively, you can set an `ORCHESTRATOR_SEED` value (a 64-character hex string) in the environment, instead of manually generating the keypair. The orchestrator will deterministically derive its key pair from this seed [8]. For example, add a line in `.env` like:

```
ORCHESTRATOR_SEED=0123456789abcdef...  (64 hex characters)
```

With a seed set, you do **not** need to provide explicit pub/secret keys; the code will generate them consistently each run from the seed. The orchestrator will still print its public key on startup [9], which should remain constant given the seed. You can take that printed key and ensure it's used as the `ORCHESTRATOR_PUBLIC_KEY` for the worker and gateway (Compose will already pass the same `ORCHESTRATOR_PUBLIC_KEY` value to all services if you define it in `.env` [10]). In practice, if you use `ORCHESTRATOR_SEED`, you might set `ORCHESTRATOR_PUBLIC_KEY` in the `.env` as well **after** you know the value, just to propagate it to the other services. The key point is that all components must share the same orchestrator public key.

After configuring one of the above, bring the stack down and up again. Now the orchestrator will use a fixed identity, and the worker will have the correct `ORCHESTRATOR_PUBLIC_KEY` from the start. This should eliminate the "wrong key" scenario. The gateway and HTTP bridge should also have this env var – in fact, the gateway will warn if `ORCHESTRATOR_PUBLIC_KEY` is not set [11]. Double-check that this warning is gone and that the gateway is forwarding requests via RPC properly.

2. Ensure Proper Startup Order (Orchestrator Before Worker):
To avoid the race condition where the worker tries to register before the orchestrator is ready, adjust your Docker Compose configuration or startup procedure:

- In the `docker-compose.yml`, add a dependency on the orchestrator for the worker service. For example:

```
worker:
  depends_on:
    orchestrator:
      condition: service_healthy
```

Ensure the orchestrator has a health check defined (in the provided compose, the orchestrator's healthcheck simply exits 0, which might not truly reflect readiness, but it at least ensures the container is up) [12] [13]. This change will make Docker Compose bring up the orchestrator first and mark it healthy before starting workers. It reduces the chance that the worker's registration call happens too early.

- If you're running services manually, start the orchestrator first, wait until it logs "Orchestrator RPC public key: ..." (indicating it's listening), then start the worker. In Compose, you can also do `docker`

`compose up -d orchestrator` first, then `docker compose up -d worker` (or bring up all but ensure orchestrator is healthy).

- **Recovery on failure:** Even with proper ordering, there could be scenarios where a worker starts slightly early or network lag causes the first registration attempt to fail. Since the current implementation doesn't retry on failure, you may need to manually restart the worker container if it came up before the orchestrator. In the long run, a more robust solution would be to implement a retry mechanism in the worker's startup (e.g. retry `register-worker` a few times with delays) or have the orchestrator periodically ping unregistered workers. For now, after applying the above fixes, monitor the logs on startup: you should see the worker log a successful registration (no warning). If you still see a failure, simply restart the worker service container; it will attempt to register again and should succeed if the orchestrator is up and the keys are correct.

By configuring a consistent orchestrator key and ordering startup, the worker will successfully connect and register with the orchestrator. The `CHANNEL_CLOSED` error should disappear. After these changes, the orchestrator's internal `workers` map will include the new worker (with its public key, model, and capabilities) [14], and the orchestrator will be able to dispatch inference requests to it. You can verify this by calling the orchestrator's `/health` or trying a sample inference request through the gateway/bridge once everything is running.

Sources: The relevant portions of the code and configuration that illustrate this solution include the worker's registration logic [1] [2], the orchestrator's key handling logic [4] [9], and the Docker Compose environment settings for orchestrator/worker [3] [10]. These show how the orchestrator's public key is meant to be provided via env and used by other services, and why a missing/mismatched key leads to the observed error.

---

[1] [2] [5] index.ts
https://github.com/alvaropaco/hyperswarm-rpc-ai-inferece-framework/blob/db13d0766ab7d68598bbc770fff1860abc76bed0/services/worker/src/index.ts

[3] [6] [10] [12] [13] docker-compose.yml
https://github.com/alvaropaco/hyperswarm-rpc-ai-inferece-framework/blob/db13d0766ab7d68598bbc770fff1860abc76bed0/docker-compose.yml

[4] [8] [9] [14] index.ts
https://github.com/alvaropaco/hyperswarm-rpc-ai-inferece-framework/blob/db13d0766ab7d68598bbc770fff1860abc76bed0/services/orchestrator/src/index.ts

[7] README.md
https://github.com/alvaropaco/hyperswarm-rpc-ai-inferece-framework/blob/db13d0766ab7d68598bbc770fff1860abc76bed0/README.md

[11] server.ts
https://github.com/alvaropaco/hyperswarm-rpc-ai-inferece-framework/blob/db13d0766ab7d68598bbc770fff1860abc76bed0/services/gateway/src/server.ts