Temporal Training Session 4: Task Queues and Workers

Table of Contents

Welcome and Overview

Welcome to Day 4. Over the past three days, we've built a solid foundation.

- On Day 1, we saw the high-level architecture of Temporal
- On Day 2, we took a deep dive into the orchestrator—the **Workflow**—and the absolute necessity of deterministic code
- On Day 3, we explored the executor—the **Activity**—and learned how to build robust, real-world logic using timeouts, retries, and idempotency

We now know *what* we want to run and *how* to make it resilient. Today, we focus on the critical link that connects them: the **Task Queue**. We'll learn how to manage, scale, and optimize the processes that do the actual work: the **Workers**. This session is all about moving from a simple "it works on my machine" setup to a design that is scalable, efficient, and production-ready.

Today's agenda, as outlined in our syllabus, will cover:

- A deep dive into what Task Queues are and how they function
- The strategic importance of separating Workflow and Activity Workers
- Techniques for scaling Workers to handle massive loads
- The Worker lifecycle and ensuring a graceful shutdown
- Best practices for observability and monitoring

A Deep Dive into Task Queues

We've mentioned Task Queues several times, but let's formalize our understanding.

A **Task Queue** is a lightweight, dynamically allocated queue that distributes tasks to workers, enabling load balancing and scalability. Its primary purpose is to decouple a task's scheduling from its execution. When a Workflow wants an Activity to run, it doesn't give the task directly to a Worker. Instead, it places a task on a Task Queue, and a Worker whose job is to listen to that specific queue will eventually pick it up and execute it.

How do Task Queues *Actually* Work?

Let's trace the lifecycle of a single Activity task:

1. **Command Creation:** Inside your Workflow code, you call `await workflow.execute_activity(...)`. The Temporal SDK doesn't execute this immediately. It translates this into a command: `ScheduleActivityTask`
2. **Command to Server:** This command is sent to the Temporal Cluster's Frontend Service
3. **History Event:** The History Service receives the command and appends an `ActivityTaskScheduled` event to the Workflow's event history. This event contains all the information needed to run the activity: the target Task Queue, the activity name, inputs, timeouts, retry policy, etc.
4. **Enqueueing:** The Matching Service takes the task and places it into the durable, in-memory Task Queue specified in the command. If the queue doesn't exist, the Server creates it on the fly
5. **Polling:** A Worker process, which has been configured to listen to this specific Task Queue, is making a long-polling request to the Matching Service, essentially asking, "Is there any work for me?"
6. **Dispatch:** The Matching Service sees the available task and the available Worker. It "matches" them, sending the task to that Worker
7. **Execution:** The Worker receives the task, executes the corresponding Activity function, and reports the result (completion or failure) back to the Temporal Server
8. **Completion Event:** The server records the result in the history (e.g., `ActivityTaskCompleted`) and schedules a `WorkflowTask` to inform the originating Workflow that its activity has finished

Key Characteristics of Task Queues

Feature	Description
Dynamic	Task Queues are created on-demand when a Workflow schedules a task for it or a Worker starts polling it. There's no need for explicit pre-registration.
Lightweight	They have very low overhead. It's common for a large system to have hundreds or even thousands of different Task Queues.
Durable	The tasks themselves are persisted. If the Matching Service node crashes, the tasks are not lost and can be recovered. If no workers are available, the tasks wait patiently in the queue.
Load Balancing	If multiple Worker processes are polling the same Task Queue, the Temporal Server automatically distributes the tasks among them, effectively balancing the load.

Workflow Tasks vs. Activity Tasks

It's important to understand that a single named Task Queue (e.g., `customer-orders`) handles **both** Workflow Tasks and Activity Tasks.

- **Workflow Tasks:** These tasks instruct a Worker to execute workflow logic. This happens when a workflow starts, an activity completes, a timer fires, or a signal is received. The Worker's job is to advance the workflow's state by running its code
- **Activity Tasks:** These tasks instruct a Worker to execute an activity function

While they share the same queue infrastructure, they are processed by different components within the Worker.

Designing Your Worker Strategy

In our simple examples so far, we've run a single worker process that handles both Workflow and Activity tasks from the same queue. This is the **Unified Worker** pattern.

```
# run_worker_unified.py
# ... imports
from .workflows import MyWorkflow
from .activities import my_activity_a, my_activity_b

# ... client setup ...
async def main():
    # A single worker polling one task queue for all tasks
    worker = Worker(
        client,
        task_queue="my-unified-queue",
        workflows=[MyWorkflow],
        activities=[my_activity_a, my_activity_b],
    )
    await worker.run()
```

This is perfectly fine for development and small applications. However, for production systems, it is highly recommended to separate them.

The Separated Worker Pattern: Why and How

The best practice is to have dedicated groups of Worker processes: one group that only handles Workflow Tasks, and one or more groups that only handle Activity Tasks.

Why Separate Your Workers?

Resource Isolation & Prioritization:

- **Workflows:** Are generally lightweight on memory but can be CPU-intensive during history replay. They must never be starved of resources, as they are the "brains" of the operation
- **Activities:** Can have vastly different resource needs. One activity might be a simple, fast API call, while another might download a 1GB file, process it in memory, and require significant CPU and RAM

The Problem: If a resource-heavy Activity runs on the same worker as a Workflow, it can consume all the CPU or memory, preventing the Workflow from making progress. This is known as the "noisy neighbor" problem. Separating them ensures that your orchestration logic is never impacted by your execution logic.

Independent Scaling:

You might find that your system needs to run thousands of concurrent activities but only has a few hundred active workflows.

By separating workers, you can scale your Activity worker fleet (e.g., run 50 pods in Kubernetes) independently from your Workflow worker fleet (which might only need 5 pods). This is far more cost-effective and efficient.

Specialized Environments & Dependencies:

- An Activity Worker might need specific Python libraries (like pandas, tensorflow, boto3) that the Workflow Worker doesn't need. This keeps the Workflow worker environment clean and lean
- An Activity Worker might need to run on a machine with a GPU, have special network access rules to reach a private database, or require specific security credentials (IAM roles) that you don't want to grant to the Workflow Worker

How to Separate Your Workers

The implementation is straightforward. You create two different worker scripts and point your `workflow.execute_activity` call to the correct Activity Task Queue.

Step 1: The Workflow Worker

This worker is lean. It only registers the Workflow definitions and polls a workflow-specific task queue.

```python
# run_workflow_worker.py
from temporalio.client import Client
from temporalio.worker import Worker
from .workflows import MyWorkflow

async def main():
    client = await Client.connect("localhost:7233")
    # This worker ONLY handles workflows. No activities are registered.
    worker = Worker(
        client,
        task_queue="my-workflow-queue",
        workflows=[MyWorkflow],
    )
    await worker.run() # This will run until shutdown
```

Step 2: The Activity Worker

This worker registers the Activity definitions and polls an activity-specific task queue.

```python
# run_activity_worker.py
from temporalio.client import Client
from temporalio.worker import Worker
from .activities import my_activity_a, my_activity_b

async def main():
    client = await Client.connect("localhost:7233")
    # This worker ONLY handles activities. No workflows are registered.
    worker = Worker(
        client,
        task_queue="my-activity-queue",
        activities=[my_activity_a, my_activity_b],
    )
    await worker.run() # This will run until shutdown
```

Step 3: Update the Workflow Definition

The final piece is to tell the Workflow where to send the activity tasks. You do this by setting the `task_queue` parameter in `workflow.execute_activity`.

```
# workflows.py
@workflow.defn
class MyWorkflow:
    @workflow.run
    async def run(self) -> str:
        # ...
        result = await workflow.execute_activity(
            my_activity_a,
            "some_input",
            task_queue="my-activity-queue", # <--- Explicitly route the
task
            start_to_close_timeout=timedelta(seconds=60),
        )
        # ...
```

By following this pattern, you gain enormous flexibility to route different types of work to specialized worker pools, creating a more robust and scalable system. You can even have multiple Activity Task Queues (e.g., `high-priority-activities`, `data-processing-activities`) polled by different sets of workers.

Scaling and Managing Workers

Horizontal Scaling

The most common way to scale is horizontally: simply run more instances of your worker process. If you have one Activity Worker handling 100 tasks per minute and your load increases to 1000 tasks per minute, you can launch nine more instances of that same worker process. The Temporal Matching Service will automatically distribute tasks across all 10 workers polling that queue. This is a powerful feature that works out of the box.

Worker Concurrency: Vertical Scaling within a Process

A single worker process is not single-threaded. It can execute multiple tasks concurrently using asyncio. You can control this concurrency with parameters on the `Worker` object.

- **max_concurrent_activities:** The maximum number of Activity Tasks that one Worker process can execute at the same time. The default is 100
- **max_concurrent_workflow_tasks:** The maximum number of Workflow Tasks that one Worker process can execute at the same time. The default is 100

```
# run_activity_worker.py
worker = Worker(
    client,
    task_queue="data-processing-activities",
    activities=[process_large_file],
    # This worker is on a powerful machine, but let's limit it
    # to running only 10 heavy activities at once to avoid OOM errors.
    max_concurrent_activities=10,
)
```

Resource-Based Scaling

You can also scale based on resource utilization:

```
# run_workflow_worker.py
worker = Worker(
    client,
    task_queue="workflow-queue",
    workflows=[MyWorkflow],
    # Workflows are CPU-intensive during replay, so limit concurrency
    # based on available CPU cores
    max_concurrent_workflow_tasks=os.cpu_count() * 2,
)
```

Worker Lifecycle and Graceful Shutdown

Starting Workers

Workers should be started as part of your application's startup process. In a containerized environment, this might be the main process in a Docker container.

```
# main.py
import asyncio
import signal
from temporalio.worker import Worker

async def main():
    # Setup client
    client = await Client.connect("localhost:7233")

    # Create worker
    worker = Worker(
        client,
        task_queue="my-queue",
        workflows=[MyWorkflow],
        activities=[my_activity],
    )

    # Handle shutdown gracefully
    async def shutdown(signal, loop):
        print(f"Received exit signal {signal.name}...")
        tasks = [t for t in asyncio.all_tasks() if t is not
asyncio.current_task()]
        [task.cancel() for task in tasks]
        await asyncio.gather(*tasks, return_exceptions=True)
        loop.stop()

    # Register signal handlers
    for sig in (signal.SIGTERM, signal.SIGINT):
        asyncio.get_event_loop().add_signal_handler(
            sig, lambda s=sig: asyncio.create_task(shutdown(s,
asyncio.get_event_loop()))
        )

    # Start the worker
    await worker.run()

if __name__ == "__main__":
    asyncio.run(main())
```

Graceful Shutdown

When a Worker receives a shutdown signal, it should:

1. **Stop accepting new tasks:** The worker stops polling for new tasks
2. **Complete in-progress tasks:** Allow currently executing tasks to complete
3. **Wait for timeouts:** Respect the configured timeouts for activities
4. **Clean up resources:** Close database connections, file handles, etc.

Temporal handles most of this automatically, but you can customize the behavior:

```
worker = Worker(
    client,
    task_queue="my-queue",
    workflows=[MyWorkflow],
    activities=[my_activity],
    # Configure shutdown behavior
    shutdown_timeout=timedelta(seconds=30),  # Wait up to 30s for tasks to
complete
)
```

Best Practices for Observability and Monitoring

Logging

Use structured logging to track worker behavior:

```python
import logging
from temporalio import activity, workflow

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

@activity.defn
async def my_activity(input_data: str) -> str:
    activity.logger.info("Starting activity", extra={
        "activity_name": "my_activity",
        "input_data": input_data,
        "worker_id": activity.info().worker_id,
    })

    try:
        result = process_data(input_data)
        activity.logger.info("Activity completed successfully", extra={
            "activity_name": "my_activity",
            "result": result,
        })
        return result
    except Exception as e:
        activity.logger.error("Activity failed", extra={
            "activity_name": "my_activity",
            "error": str(e),
        })
        raise
```

Metrics

Track key metrics for worker performance:

```python
from temporalio import activity
import time

@activity.defn
async def monitored_activity(input_data: str) -> str:
    start_time = time.time()

    try:
        result = await process_data(input_data)
        # Record success metrics
        duration = time.time() - start_time
        record_metric("activity_duration_seconds", duration, {
            "activity_name": "monitored_activity",
            "status": "success"
        })

        return result
    except Exception as e:
        # Record failure metrics
        duration = time.time() - start_time
        record_metric("activity_duration_seconds", duration, {
            "activity_name": "monitored_activity",
            "status": "failure",
            "error_type": type(e).__name__
        })
        raise
```

Health Checks

Implement health checks for your workers:

```
from temporalio import activity
import asyncio

@activity.defn
async def health_check() -> dict:
    """Activity that performs health checks on the worker."""
    health_status = {
        "worker_id": activity.info().worker_id,
        "timestamp": time.time(),
        "checks": {}
    }

    # Check database connectivity
    try:
        await check_database_connection()
        health_status["checks"]["database"] = "healthy"
    except Exception as e:
        health_status["checks"]["database"] = f"unhealthy: {str(e)}"

    # Check external API connectivity
    try:
        await check_external_api()
        health_status["checks"]["external_api"] = "healthy"
    except Exception as e:
        health_status["checks"]["external_api"] = f"unhealthy: {str(e)}"

    return health_status
```

Production Deployment Considerations

Containerization

Use Docker to containerize your workers:

```
# Dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy application code
COPY . .

# Run the worker
CMD ["python", "run_worker.py"]
```

Kubernetes Deployment

Deploy workers as Kubernetes deployments:

```
# worker-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: temporal-worker
spec:
  replicas: 3
  selector:
    matchLabels:
      app: temporal-worker
  template:
    metadata:
      labels:
        app: temporal-worker
    spec:
      containers:
      - name: worker
        image: myapp/worker:latest
        env:
        - name: TEMPORAL_HOST
          value: "temporal-server:7233"
        - name: TASK_QUEUE
          value: "my-queue"
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
          limits:
            memory: "512Mi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 5
```

Auto-scaling

Configure auto-scaling based on queue depth:

```
# hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: temporal-worker-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: temporal-worker
  minReplicas: 2
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

Summary

Today we've covered the essential aspects of managing Task Queues and Workers in production:

1. **Task Queue Architecture:** Understanding how tasks flow through the system
2. **Worker Separation:** Why and how to separate workflow and activity workers
3. **Scaling Strategies:** Horizontal and vertical scaling techniques
4. **Lifecycle Management:** Proper startup and shutdown procedures
5. **Observability:** Logging, metrics, and health checks
6. **Production Deployment:** Containerization and orchestration considerations

These concepts are crucial for building scalable, maintainable Temporal applications that can handle real-world production loads.

The combination of proper worker design, effective scaling strategies, and comprehensive monitoring will ensure your Temporal workflows run reliably and efficiently in any environment.