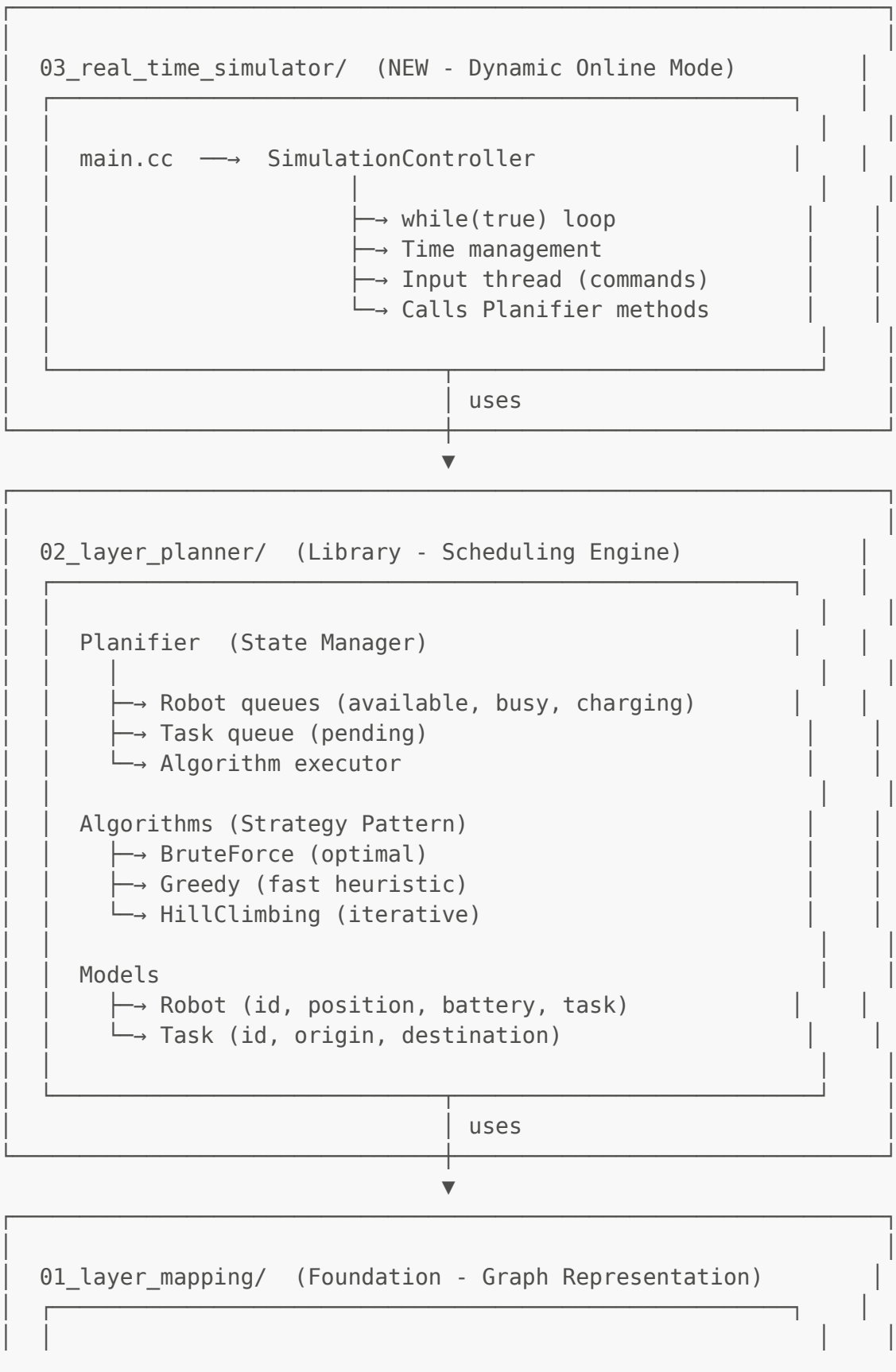
System Architecture Overview

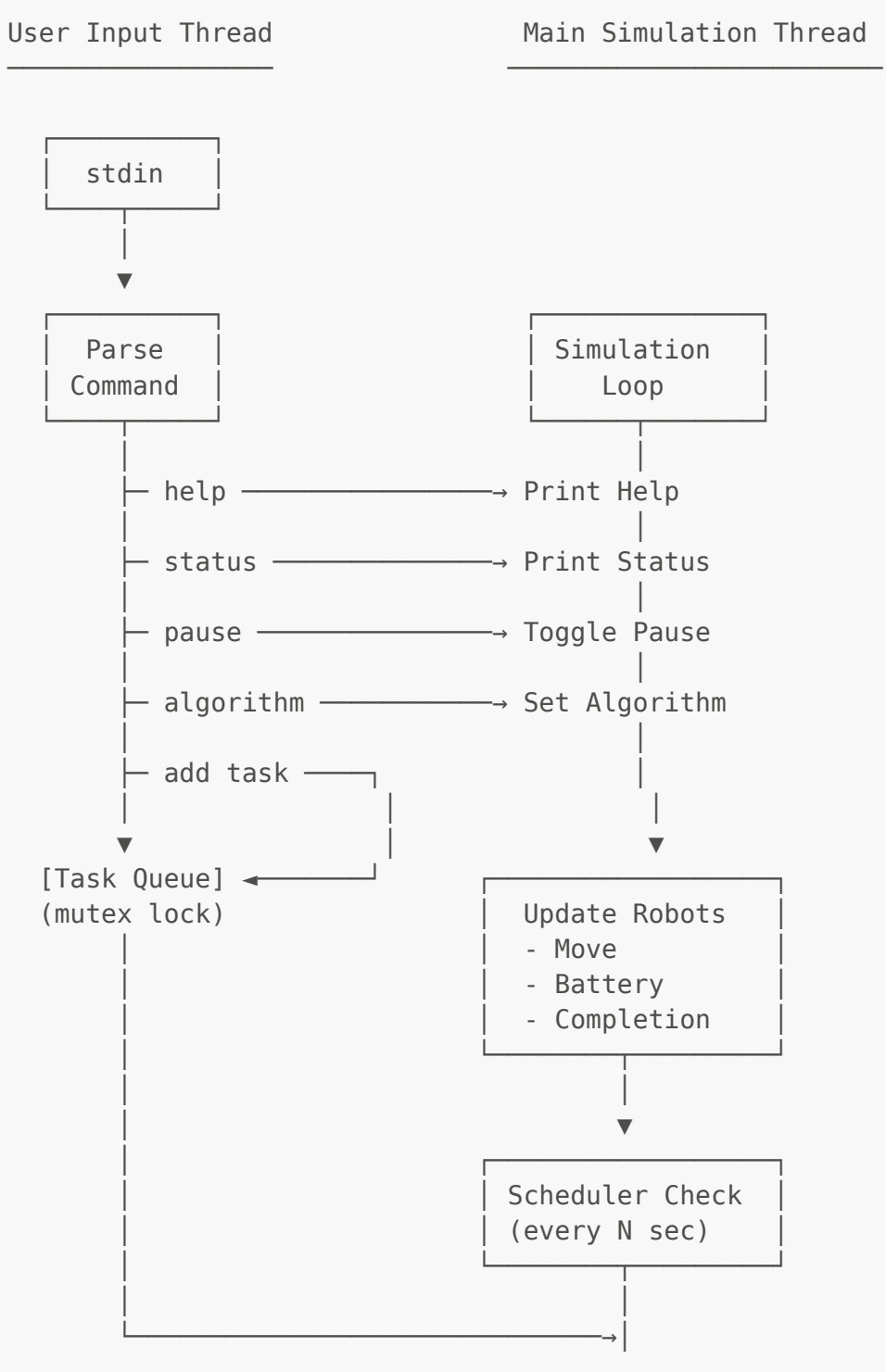
Layer Structure



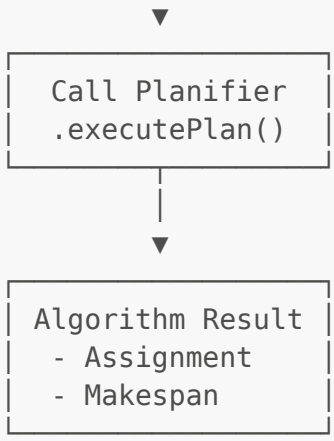
PROF

```
|   |  Graph                                                    |     |
|   |     ├─→ Nodes (waypoints, charging, pickup, dropoff)      |     |
|   |     ├─→ Edges (connections, distances, speeds)            |     |
|   |     ├─→ Shortest path algorithms                          |     |
|   |     └─→ File I/O (load/save)                              |     |
|   |                                                           |     |
|   └───────────────────────────────────────────────────────────┘     |
|                                                                     |
└─────────────────────────────────────────────────────────────────────┘
```

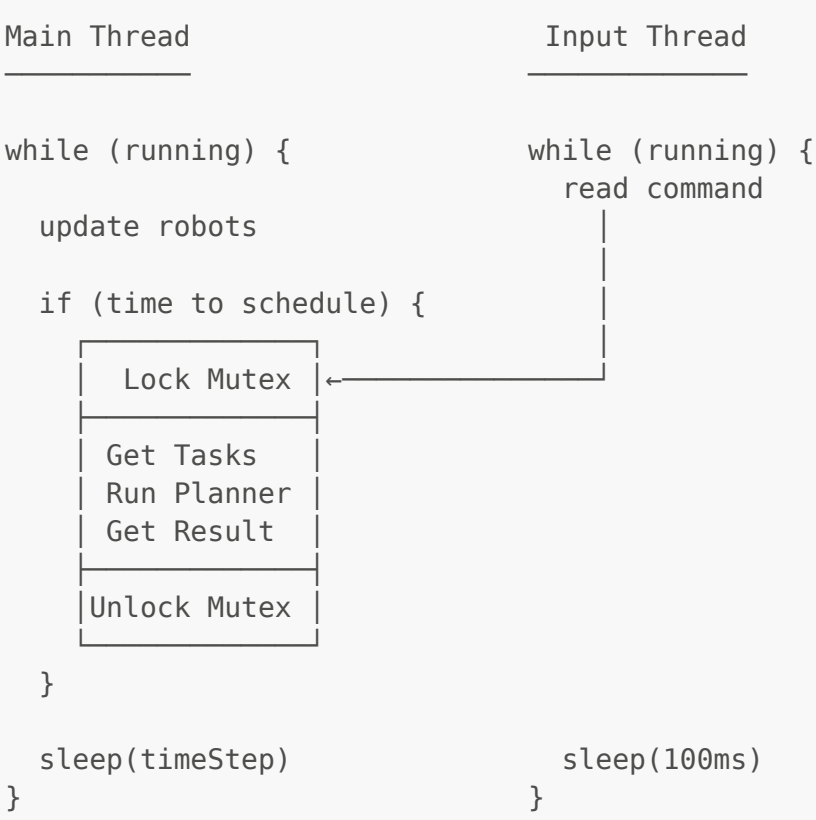
Real-Time Simulator Data Flow



PROF



Thread Safety



Key Design Decisions

1. Non-Invasive Integration

- Real-time simulator is a **new application**
- Planifier is used as a **library**
- No modifications to existing 02_layer_planner code
- Original main.cc remains untouched

2. Separation of Concerns

PROF

- **Layer 1**: Graph structure and pathfinding
- **Layer 2**: Task scheduling and optimization
- **Layer 3**: Real-time execution and interactivity

3. Strategy Pattern

- Algorithms are interchangeable
- Can switch during runtime
- Each algorithm implements same interface

4. Thread-Safe Design

- Mutex protects shared state
- Lock only during critical sections
- Non-blocking input handling

File Dependencies

```
real_time_simulator/
├── main.cc
│   └─→ requires: SimulationController.hh, Graph.hh, Task.hh
│
├── src/SimulationController.cc
│   └─→ requires: SimulationController.hh
│                 Planifier.hh
│                 algorithms/*.hh
│                 Robot.hh, Task.hh, Graph.hh
│
└── include/SimulationController.hh
    └─→ requires: Planifier.hh, Graph.hh, Task.hh, Robot.hh
```

PROF

Compilation Chain

```
Source Files                Object Files              Executable
────────────                ────────────              ──────────

main.cc                 ─→ main.o                  ┐
SimulationController.cc ─→ SimulationController.o  │
                                                   │
Planifier.cc            ─→ Planifier.o             │
Robot.cc                ─→ Robot.o                 │
Task.cc                 ─→ Task.o                  ├─→ simulator
01_BruteForce.cc        ─→ 01_BruteForce.o         │
02_Greedy.cc            ─→ 02_Greedy.o             │
03_HillClimbing.cc      ─→ 03_HillClimbing.o       │
SchedulerUtils.cc       ─→ SchedulerUtils.o        │
TSPSolver.cc            ─→ TSPSolver.o             │
AssignmentPrinter.cc    ─→ AssignmentPrinter.o     │
```

```
                                                    │
Graph.cc                —→ Graph.o                  ┘
```

Usage Flow

```
1. User launches:
   ./build/simulator <graphId> [numRobots] [testCase]

2. main.cc:
   - Parses arguments
   - Loads Graph from file
   - Loads initial Tasks (if testCase provided)
   - Creates SimulationController

3. SimulationController.start():
   - Spawns input thread
   - Enters main loop

4. Main Loop (continuous):
   while (running) {
     - Update simulation time
     - Update robots (position, battery, tasks)
     - Check if scheduler interval elapsed
     - If yes: run scheduler cycle
     - Sleep for timeStep
   }

5. Input Thread (parallel):
   while (running) {
     - Wait for user input
     - Parse command
     - Execute action (add task, change algo, etc.)
     - Continue
   }

6. Scheduler Cycle (periodic):
   - Lock task queue
   - Call Planifier.executePlan()
   - Algorithm runs and returns result
   - Assignment printed
   - Unlock

7. User quits:
   - Sets running = false
   - Main loop exits
   - Input thread joins
   - Cleanup and exit
```

PROF

Extension Points

To add new features to the simulator:

1. Add New Commands

- Update `processCommand()` method
- Add case to command parser
- Implement command handler

2. Add Robot Behaviors

- Implement in `updateRobots()`
- Access robot queues via Planifier
- Update positions, battery, states

3. Add Visualization

- Create new class: `SimulationVisualizer`
- Hook into main loop
- Render state at each step

4. Add Metrics/Logging

- Create new class: `SimulationLogger`
- Record events and statistics
- Export to file or database

5. Add Network Interface

- Create new class: `SimulationServer`
- Expose REST API or WebSocket
- Allow remote monitoring/control

PROF

Note: All extensions should be added to the real_time_simulator layer, keeping the existing layers (01 and 02) unchanged.