The crux of the hybrid system lies in combining the global search capabilities of FPA with the local search and convergence properties of SA. The Hybrid Optimization Mechanism aims to achieve a balance between exploration and exploitation in the search space.

Exploration: The capacity to venture far into the search to identify entirely new and possibly better solutions. This is primarily contributed by FPA's global pollination feature.

Exploitation: The capacity to fine-tune existing solutions to reach their optimal state. This is contributed by SA's acceptance probability and cooling schedule.

During the optimization process, for each flower in the population, FPA performs either a global or local pollination to generate a new position (`new_position`). This new position represents a candidate solution, and its fitness (`new_fitness`) is evaluated using the problem's fitness function.

Instead of immediately accepting this new position, the algorithm asks Simulated Annealing (SA) whether to accept it or not. This is done using the `accept(new_fitness)` method of the SA class.

Fitness Function

The algorithm utilizes a custom fitness function tailored for the specific optimization problem. This function evaluates the quality of a given solution in the search space.

The fitness function incorporates multiple factors:

Makespan: The total time required to complete all tasks.
Cost: The sum of the costs associated with each server.
Utilization: Average utilization of all servers.

The fitness value (`new_fitness`) serves as a measure for evaluating candidate solutions, and it guides the decision-making process in both the FPA and SA stages.

Initialization
Initialize a flower population with random positions and fitness values and set up Simulated Annealing parameters, particularly the initial temperature.

```
Initialize SA parameters, most importantly the temperature.
  def __init__(self, fitness_function, initial_solution,
  initial_temperature,
  final_temperature, cooling_rate):
    self.fitness_function = fitness_function
    self.current_solution = initial_solution
    self.current_fitness = self.fitness_function(self.current_solution)
    self.best_solution = self.current_solution
    self.best_fitness = self.current_fitness
    self.temperature = initial_temperature
    self.final_temperature = final_temperature
    self.cooling_rate = cooling_rate
```

- **Pollination Stage**:

For each flower in the population, FPA generates a new candidate position (`new_position`).

FPA calculates the fitness (`new_fitness`) of this new position using a predefined fitness function relevant to the optimization problem.

```
def __init__(self, fitness_function, population_size=100, decision_variables=10,
lower_bound=0, upper_bound=1, generations=1000, switch_prob=0.8, gamma=0.1,
beta=1.5):
    self.population = [Flower(
    decision_variables, lower_bound, upper_bound) for _ in range(population_size)]
    self.global_best = None
    self.fitness_function = fitness_function
    self.population_size = population_size
    self.decision_variables = decision_variables
    self.lower_bound = lower_bound
    self.upper_bound = upper_bound
    self.generations = generations
    self.switch_prob = switch_prob
    self.gamma = gamma
    self.beta = beta


def global_pollination(self, flower):
    step_size = self.gamma * self.levy_flight(self.beta)
    new_position = flower.position + step_size * \
    (flower.position - self.global_best.position)
    return new_position

def local_pollination(self, flower):
    flower_j = np.random.choice(self.population)
    epsilon = np.random.uniform(
    low=0, high=1, size=self.decision_variables)
    new_position = flower.position + epsilon * \
    (flower_j.position - flower.position)
    return new_position
```

SA Filtering

Before adopting the `new_position` and its associated `new_fitness`, the hybrid mechanism consults the Simulated Annealing algorithm.

SA computes the acceptance probability (`probability = np.exp(delta / temperature)`) based on the difference in fitness (`delta`) between the current position and the new candidate position, as well as the current temperature.

SA then makes a probabilistic decision to either accept or reject the `new_position`. If accepted, the flower's attributes are updated; otherwise, they remain unchanged.

```
def optimize(self):
    self.global_best = self.population[0]
    self.global_best.fitness = self.fitness_function(self.global_best.position)
    sa = SimulatedAnnealing(self.fitness_function, self.global_best.position,
    initial_temperature=10, final_temperature=0.001, cooling_rate=0.9)

    for g in range(self.generations):
        for flower in self.population:
        flower.fitness = self.fitness_function(flower.position)
        self.find_global_best()

        new_position = self.pollination(flower)
        new_fitness = self.fitness_function(new_position)

        if sa.accept(new_fitness):
            flower.position = new_position
            flower.fitness = new_fitness
```

Cooling Schedule

After the SA Filtering stage is complete for all flowers in the population, the temperature parameter for SA is reduced according to a cooling schedule. This makes it increasingly hard to accept worse solutions, forcing the algorithm to converge.

```
sa.cool_down() # cool down the temperature after each generation
self.find_global_best()
```

Conclusion

By combining these two algorithms, the hybrid approach seeks to achieve a balance between exploring new, potentially better solutions (global search by FPA) and refining current solutions (local search by SA), offering the following advantages.

Global-Local Synergy: FPA's global and local pollination ensure broad exploration, while SA's probabilistic acceptance and cooling schedule allow fine-tuning of solutions.

Dynamic Adaptation: The probabilistic elements in both FPA and SA, coupled with the cooling schedule and switching mechanism, make the hybrid algorithm highly adaptable to complex landscapes.

Reduced Likelihood of Stalling: The hybrid nature makes it less likely for the algorithm to get stuck in local minima, a common issue in optimization problems.