Pseudo code:

```
Create a class called Philosopher with properties name and state
Philosopher class has methods take_chopsticks() and release_chopsticks()
    take_chopsticks(waiter):
        If both left and right chopsticks exist and philosopher is hungry:
                If both chopsticks are available and waiter.grant_permission() returns True:
                        Set philosopher state to eating
                        Set both chopsticks state to unavailable
    release_chopsticks(waiter):
        Set philosopher state to thinking
        Set both chopsticks state to available
r node and calls take_chopsticks() when the philosopher is hungry
Initialize ZooKeeper client
Create 10 chopstick nodes with initial state available
Create 5 philosopher nodes with initial state thinking and watch them for changes using
philosopher_watch_func
Create a Waiter object
Loop indefinitely, sleeping for 1 second at each iteration
for each philosopher node, call take_chopsticks(waiter)
    call release_chopsticks(waiter) when philosopher is done eating
```

In this code, the take_chopsticks() method takes a waiter parameter, which represents the waiter responsible for managing access to the chopsticks. Before allowing the philosopher to start eating, the method first requests permission from the waiter by calling waiter.grant_permission(). If the permission is granted (i.e., the method returns True), then the philosopher can proceed to take both chopsticks and start eating.

Similarly, the release_chopsticks() method also takes a waiter parameter, as it needs to inform the waiter that the chopsticks have been released and are available for other philosophers to use.

At each iteration of the main loop, the program calls take_chopsticks() on each philosopher node that is currently in the "hungry" state, passing in the waiter object. If a philosopher is done eating, the program calls release_chopsticks() on the corresponding node to release the chopsticks and change the philosopher's state to "thinking".

1. *Initialize ZooKeeper client and establish a connection.*
2. *Create a persistent znode (or node) for each philosopher in ZooKeeper to represent their state, with the "thinking" value set as the initial data, and with the SEQUENTIAL flag set.*
3. *Each philosopher will have three states: "thinking", "hungry" and "eating" which are represented by different values in their respective znode.*
4. *Create an ephemeral znode for the waiter in ZooKeeper to indicate whether the philosopher can eat or not, with an initial value of "available".*
5. *When a philosopher wants to eat, they check if they are hungry and if the two chopsticks on either side of them are available.*
6. *If both chopsticks are available and the waiter permits it by setting the value of its znode to "unavailable", they change their state to "eating".*
7. *If one or more of the chopsticks are not available or the waiter does not permit it, they remain in the "hungry" state.*
8. *Once a philosopher finish eating, they release the chopsticks and change their state back to "thinking".*
9. *Whenever a philosopher changes their state, they update their respective znode in ZooKeeper using the setData operation.*
10. *The waiter sets a watch on all philosopher znodes to listen for changes in their state. Whenever a philosopher changes their state to "hungry", the waiter checks if both chopsticks are available.*
11. *If both chopsticks are available, the waiter grants permission to the philosopher by changing the value of the philosopher's znode to "eating" using the setData operation.*
12. *If both chopsticks are not available, the waiter denies permission by leaving the philosopher's znode as "hungry".*

This algorithm uses Zookeeper's coordination service to ensure that only one philosopher is eating at a time, and it avoids deadlocks by having the waiter act as a central authority to grant permission to the philosophers to eat.

Explanation on each Step:

1. Initialize Zookeeper client and establish a connection:
   - The first step is to initialize a Zookeeper client and establish a connection with the Zookeeper server. This step is necessary because all communication between the philosophers and the waiter will happen through Zookeeper. By establishing a connection with the Zookeeper server, the client can access the necessary resources (such as znodes) in Zookeeper.

2. Create a znode (or node) for each philosopher in Zookeeper to represent their state:
   - Next, you need to create a znode (or node) in Zookeeper for each philosopher participating in the dining philosophers problem. Each znode will represent the state of the corresponding philosopher. This step is necessary because it allows the philosophers to communicate their state with each other and with the waiter.

3. Each philosopher will have three states: "thinking", "hungry" and "eating" which are represented by different values in their respective znode:
    - Each philosopher will have three possible states - "thinking", "hungry" and "eating". These states will be represented by different values in the corresponding philosopher's znode in Zookeeper. This step is necessary because it allows the philosophers to indicate their current state to each other and to the waiter.

4. The waiter also has a znode to indicate whether the philosopher can eat or not:
    - The waiter, who is responsible for managing access to the chopsticks, will have a separate znode in Zookeeper to indicate whether a particular philosopher can eat or not. This step is necessary because it allows the waiter to control access to the chopsticks and prevent multiple philosophers from trying to grab the same chopstick at the same time.

5. When a philosopher wants to eat, they check if they are hungry and if the two chopsticks on either side of them are available:
    - When a philosopher wants to eat, they first check if they are currently in the "hungry" state. If yes, they then check if the two chopsticks on either side of them are available or not. This step is necessary because it ensures that a philosopher can only attempt to eat when they are actually hungry and when the required resources (the two chopsticks) are available.

6. If both chopsticks are available and the waiter permits it, they change their state to "eating":
    - If both chopsticks are available and the waiter grants permission, the philosopher changes their state to "eating". This step is necessary because it allows the philosopher to start eating and prevents other philosophers from trying to grab the same chopsticks at the same time.

7. If one or more of the chopsticks are not available or the waiter does not permit it, they remain in the "hungry" state:
    - If one or more of the chopsticks are not available or the waiter does not grant permission, the philosopher remains in the "hungry" state. This step is necessary because it prevents philosophers from attempting to eat when the required resources are not available or when another philosopher is already using them.

8. Once a philosopher finishes eating, they release the chopsticks and change their state back to "thinking":
    - Once a philosopher finishes eating, they release the chopsticks and change their state back to "thinking". This step is necessary because it allows other philosophers to use the chopsticks and prevents a single philosopher from holding onto them for too long.

9. Whenever a philosopher changes their state, they update their respective znode in Zookeeper:

- Whenever a philosopher changes their state, they need to update their corresponding znode in Zookeeper to reflect the new state. This step is necessary because it allows other philosophers and the waiter to know the current state of each philosopher and respond accordingly.

10. The waiter listens for changes in the philosophers' znodes, and whenever a philosopher changes their state to "hungry", the waiter checks if both chopsticks are available:
    - The waiter needs to listen for updates to the philosophers' znodes in Zookeeper. Whenever a philosopher changes their state to "hungry", the waiter checks if both chopsticks on either side of the philosopher are available or not. This step is necessary because it allows the waiter to determine whether a philosopher can eat or not based on the availability of the required resources.

11. If both chopsticks are available, the waiter grants permission to the philosopher by changing the value of the philosopher's znode to "eating":
    - If both chopsticks are available, the waiter grants permission to the philosopher by updating the philosopher's znode in Zookeeper with the new state of "eating". This step is necessary because it allows the philosopher to start eating and prevents other philosophers from trying to grab the same chopsticks at the same time

12. If both chopsticks are not available, the waiter denies permission by leaving the philosopher's znode as "hungry". In this case, the philosopher's znode in Zookeeper is left unchanged and continues to reflect the "hungry" state.

Use of Zookeeper's API in the Pseudo code and in the bonus implementation:

1. Import the Kazoo library to use ZooKeeper's API.
2. Create a client object and connect it to the ZooKeeper instance using the KazooClient() function.
3. Use the *create()* function of the client object to create a node in the ZooKeeper hierarchy. The *create()* function takes two arguments: the path to the new node and the data to be stored in the node.
4. Use the *get()* function of the client object to retrieve the data stored in a node. The *get()* function takes one argument: the path to the node to be retrieved.
5. Use the *set()* function of the client object to update the data stored in a node. The *set()* function takes two arguments: the path to the node to be updated and the new data to be stored in the node.
6. Use the ensure_path() function of the client object to create a path if it doesn't already exist. This function ensures that the specified path and any intermediate paths are created in the ZooKeeper hierarchy. The ensure_path() function takes one argument: the path of the node to be created.

This implementation prevent deadlocks by using a centralized approach where the waiter is responsible for managing access to the chopsticks.

In this approach, each philosopher can only attempt to eat when they are actually hungry and when both required chopsticks are available. The philosopher will change its state to "eating" if both chopsticks are available and the waiter permits it by setting its corresponding znode in Zookeeper with the value of "unavailable". If one or more chopsticks are not available or the waiter does not permit it, the philosopher remains in the "hungry" state.

Furthermore, the waiter sets a watch on all philosopher znodes to listen for changes in their state. Whenever a philosopher changes their state to "hungry", the waiter checks if both chopsticks are available. If both chopsticks are available, the waiter grants permission to the philosopher by updating the philosopher's znode in Zookeeper with the new state of "eating". If both chopsticks are not available, the waiter denies permission by leaving the philosopher's znode as "hungry".

This approach ensures that a philosopher can only start eating when both chopsticks are available and the waiter grants permission. Therefore, there is no possibility of a deadlock occurring where multiple philosophers are holding onto one or more chopsticks and unable to proceed.

Code Implementation:

Threading is relevant in the code because it allows the philosophers to execute their actions concurrently. Each philosopher is represented by a separate thread, allowing them to perform their actions simultaneously. Without threading, the philosophers would have to take turns in executing their actions, which would not accurately simulate the dining philosophers problem.

In the code, the philosopher_thread_func function is executed as a separate thread for each philosopher. This allows each philosopher to continuously alternate between thinking and hungry states, attempt to take chopsticks, eat for a random amount of time, release the chopsticks, and go back to thinking. The use of threads ensures that the actions of different philosophers can happen concurrently, which is essential for simulating the concurrent nature of the problem.

Additionally, there is a separate thread for watching the state of the waiter and the philosophers. These threads monitor changes in the ZooKeeper nodes and invoke the corresponding functions (waiter_watch_func and philosopher_watch_func) when changes occur. The use of threads for watching these nodes allows for asynchronous notifications and immediate response to state changes, ensuring that the philosophers can attempt to take chopsticks as soon as they become available.

Overall, threading enables concurrent execution and responsiveness in simulating the dining philosophers problem.