Student Name: Iliadis Viktoras
Student ID: 8180026

Big Data Management Systems

Assignment 3 - Redis / Key-Value Stores

This report presents the work undertaken for the third assignment of the BDMS course during the summer semester of 2023, offered by the Department of Management Science and Technology at AUEB. The objective of this assignment was to design and implement a meeting management system using Redis and Python.

The developed system enables administrators to create meetings, handle various instances of these meetings, monitor participant activities, and facilitate communication through chat messages. The report provides a concise overview of the system's schema and highlights the key scripts that were developed to deliver the necessary functionality. To access the complete source code for this assignment, please refer to the following repository.

System Schema

The Meeting Management System utilizes a Redis database to store and manage data related to users, meetings, meeting instances, events, and chat messages. The schema consists of the following key components:

- **Users:** Stores user information such as user ID, name, age, gender, and email.
- **Meetings:** Stores details about meetings, including meeting ID, title, description, public status, and audience (if applicable).
- **Meeting Instances:** Stores specific instances of meetings, including order ID, start time, end time, and activation status.
- **Events Log:** Logs user events such as joining a meeting, leaving a meeting, or timing out. Each event record contains information such as event ID, user ID, event type, and timestamp.
- **Chat Messages:** Stores chat messages exchanged during meetings, including message ID, user ID, message content, and timestamp.

All the key/value pairs that were stored followed a simple naming convention. For example, the key for a user record and for a meeting record is:

`user:{user_id} and meeting:{meeting_id}`

The only exception to this convention is the key for the events log when logging `"join_meeting"` and `"leave_meeting"` events. In these cases in order to be able to use patterns to query them optimally, the key is:

`event:{event_id}:userID:{userID}:meetingInstanceID:{meetingInstanceID}:event_type:{eventType}`

As required, two databases were used. Database with index 0 that includes all the user, meeting, meeting instance, and chat message records. Database with index 1 that includes all the events log records.

Functionality Implementation

Overview

To implement the Meeting Management System, I developed two key scripts: scheduler.py, and program.py.

scheduler.py: This script handles the activation and deactivation of meeting instances based on their scheduled start and end times. It scans the meeting instances every minute and updates their activation status accordingly. The scheduler ensures that active meetings are made available at the scheduled start time and deactivated once the scheduled end time is reached.

program.py: This script serves as the main program for the Meeting Management System. It provides a set of functions to interact with the Redis database and perform various operations. These functions include joining and leaving meetings, displaying participants, showing active meetings, posting chat messages, retrieving chat messages, displaying participant join timestamps, and fetching chat messages for a specific user in an active meeting.

The combination of these scripts enables the Meeting Management System to effectively manage meetings, track participant activities, and facilitate real-time communication among users.

In order to better demonstrate the system, two helper scripts named `redis_init.py` and `printRedis.py` were also created. The first script initializes the Redis database with sample data, and the second prints the contents of the Redis database.

```
!python redis_init.py

print("\n---------------------------------------\n")

!python printRedis.py
```

```
Redis initialized with sample data.

---------------------------------------

Users:
User ID: 1
    userID: 1
    name: John Doe
    age: 25
    gender: Male
    email: john.doe@example.com

User ID: 2
    userID: 2
    name: Jane Smith
    age: 30
    gender: Female
    email: jane.smith@example.com

Meetings:
Meeting ID: 1
    meetingID: 1
    title: Team Standup
    description: Daily standup meeting for the team
    isPublic: True
...
    message: Hi, John!
    timestamp: 2023-05-08T10:12:00
```

Core functionality

scheduler.py The scheduler.py script is responsible for activating and deactivating meeting instances based on their scheduled start and end times. It scans the meeting instances every minute and updates their activation status accordingly.

Below I demonstrate the functionality by running the scheduler script for 3 minutes.

```
Wed May 10 23:49:20 2023
Deactivating meeting instance:  2
Scan finished.

Wed May 10 23:50:20 2023
Scan finished.

Wed May 10 23:51:20 2023
Scan finished.

^C
Traceback (most recent call last):
  File "/home/mainuser/Project3/scheduler.py", line 48, in <module>
    scheduler()
  File "/home/mainuser/Project3/scheduler.py", line 44, in scheduler
    time.sleep(60)
KeyboardInterrupt
```

The **program.py**, contains all the functions that were required by the assignment. For each function, a brief description will be provided, followed by the code and the various use cases.

"A user joins an active meeting instance". The `join_meeting` function is responsible for allowing a user to join a meeting. It performs the following steps:

- It first checks if the provided `user_id` is valid by retrieving the user's information from the Redis database. If the user doesn't exist, it returns an "Invalid user ID" message.
- It then retrieves the meeting instance information based on the provided `meeting_instance_id` . If the meeting instance doesn't exist, it returns an "Invalid meeting instance ID" message.
- If the meeting instance is not active it returns a "Meeting instance is not active" message.
- It also checks if the meeting is public or if the user's email is in the audience list. If the user is allowed to join the meeting, it proceeds with the next steps. Otherwise, it returns a message stating that the user is not allowed to join the meeting.
- It checks if the user has already joined the meeting by calling the `check_user_in_meeting function`. If the user is already in the meeting, it returns a message stating that the user is already in the meeting.
- If the user is eligible to join the meeting, it updates the event log in the Redis database. It generates a new event ID and creates a new event entry with the necessary information, including the user ID, meeting instance ID, event type (`"join_meeting"`), and timestamp.

Finally, it returns a success message indicating that the user has successfully joined the meeting.

```python
def join_meeting(user_id, meeting_instance_id):
    # select the main database
    redis_db.select(0)

    user = redis_db.hgetall(f"user:{user_id}")
    if not user:
        return "Invalid user ID"

    user_email = user["email"]

    meeting_instance = redis_db.hgetall(f"meeting_instance:{meeting_instance_id}")

    if not meeting_instance:
        return "Invalid meeting instance ID"

    # check if the meeting instance is active
    is_active = meeting_instance["isActive"] == "True"
    if not is_active:
        return "Meeting instance is not active"

    meeting_id = meeting_instance["meetingID"]
    meeting = redis_db.hgetall(f"meeting:{meeting_id}")
    if not meeting:
        return "Invalid meeting ID"

    is_public = meeting["isPublic"] == "True"
    audience = json.loads(meeting["audience"])

    if is_public or user_email in audience:
        # Check if the user has joined the meeting
        if check_user_in_meeting(user_id, meeting_instance_id):
            return "User is already in the meeting"

        # Update event log
        redis_db.select(1)
        event_id = redis_db.incr("event_id")
        event = {
            "eventID": event_id,
            "userID": user_id,
            "meetingInstanceID": meeting_instance_id,
            "event_type": "join_meeting",
            "timestamp": datetime.now().isoformat(),
        }

        redis_db.hmset(
            f"event:{event_id}:userID:{user_id}:meetingInstanceID:{meeting_instance_id}:
    event_type:join_meeting",
            event,
        )
        # select the main database
        redis_db.select(0)

        return f"User {user_id} successfully joined meeting"

    return f"User {user_id} not allowed to join the meeting"
```

Listing 1: join meeting Function

```
def check_user_in_meeting(user_id, meeting_instance_id):
    # select the main database
    redis_db.select(1)
    event_id = redis_db.get("event_id")

    # Check if the user has joined the meeting and is still in the meeting
    events = redis_db.keys(
        f"event:*:userID:{user_id}:meetingInstanceID:{meeting_instance_id}:event_type:
    join_meeting"
    )

    if not events:
        return False

    for event_key in events:
        keys = redis_db.keys(
            f"event:*:userID:{user_id}:meetingInstanceID:{meeting_instance_id}:event_type
    :*"
        )
        userEvents = []
        for key in keys:
            userEvents.append(redis_db.hgetall(key))

        # sort the events by timestamp
        userEvents.sort(key=lambda x: x["timestamp"])
        if userEvents[-1]["event_type"] == "join_meeting":
            return True

    return False
```

Listing 2: check user in meeting Function

Here's a demonstration of the function:

```
# User 1 joins meeting instance 2
print(program.join_meeting(1, 2))
 # User 1 tries to join meeting instance 2 again
print(program.join_meeting(1, 2))
```

```
# User 2 tries to join meeting instance 2 without permission
print(program.join_meeting(2, 2))
# User 1 tries to join meeting instance 1 which is not active
print(program.join_meeting(1, 1))
# User 3 (that doesn't exist) tries to join meeting instance 2
print(program.join_meeting(3, 2))
```

```
User 1 successfully joined meeting
User is already in the meeting
User 2 not allowed to join the meeting
Meeting instance is not active
Invalid user ID
```

"A user leaves a meeting that has joined". The `leave_meeting` function is responsible for allowing a user to leave a meeting. It performs the following steps:

- It checks if the user has already joined the meeting by calling the `check_user_in_meeting` function. If the user is not in the meeting, it returns a message stating that the user is not in the meeting.
- If the user is in the meeting, it updates the event log in the Redis database. It generates a new event ID and creates a new event entry with the necessary information, including the user ID, meeting instance ID, event type (`"leave_meeting"`), and timestamp.
- Finally, it returns a success message indicating that the user has successfully left the meeting.

```python
def leave_meeting(user_id, meeting_instance_id):
    # select the main database
    redis_db.select(1)
    event_id = redis_db.get("event_id")

    # Check if the user has joined the meeting
    # Get the event log
    joined_arr = redis_db.keys(
        f"event:*:userID:{user_id}:meetingInstanceID:{meeting_instance_id}:event_type:join_meeting"
    )

    if joined_arr:
        still_online = check_user_in_meeting(user_id, meeting_instance_id)
        if still_online == False:
            return "User has already left the meeting"

        # Update event log
        redis_db.select(1)
        event_id = redis_db.incr("event_id")
        event = {
            "eventID": event_id,
            "userID": user_id,
            "meetingInstanceID": meeting_instance_id,
            "event_type": "leave_meeting",
            "timestamp": datetime.now().isoformat(),
        }
        redis_db.hmset(
            f"event:{event_id}:userID:{user_id}:meetingInstanceID:{meeting_instance_id}:event_type:leave_meeting",
            event,
        )
        redis_db.select(0)

        return "User successfully left the meeting"

    return "User has not joined the meeting"
```

Listing 3: leave meeting Function

Here's a demonstration of the `leave_meeting` function:

```
# User 1 leaves meeting instance 2
print(program.leave_meeting(1, 2))
 # User 2 tries to leave meeting instance 1 that they have not joined
print(program.leave_meeting(2, 1))
# User 1 tries to leave meeting instance 2 again
```

```
print(program.leave_meeting(1, 1))

User successfully left the meeting
User has not joined the meeting
User has not joined the meeting
```

"Show meeting's current participants".The `show_current_participants` function is responsible for displaying the current participants in a meeting. It performs the following steps:

- It first retrieves the meeting instance information based on the provided `meeting_instance_id`. If the meeting instance doesn't exist, it returns an "Invalid meeting instance ID" message.
- Then, it retrieves the list of participants from the Redis database based on the meeting instance ID.
- Finally, it returns the list of participants.

```python
def show_current_participants(meeting_instance_id):
    # select the main database
    redis_db.select(0)
    participants = []

    meeting_instance = redis_db.keys(f"meeting_instance:{meeting_instance_id}")
    if meeting_instance == []:
        return "Invalid meeting instance ID"

    # select the event database
    redis_db.select(1)

    events = redis_db.keys(
        f"event:*:userID:*:meetingInstanceID:{meeting_instance_id}:event_type:join_meeting
    "
    )

    usersChecked = {}

    for event_key in events:
        participant_id = int(event_key.split(":")[3])
        if participant_id in usersChecked:
            continue
        usersChecked[participant_id] = True

        # check if participant has left the meeting
        user_still_in_meeting = check_user_in_meeting(
            participant_id, meeting_instance_id
        )
        if user_still_in_meeting:
            participants.append(participant_id)

    return participants
```

Listing 4: Show current participants function

Here's a demonstration of the `show_current_participants` function:

```
# Show participants of meeting instance 1, should be empty
print("Meeting instance 1 participants: ", program.show_current_participants(1))
# User 1 joins meeting instance 2
print(program.join_meeting(1, 2))
# Show participants of meeting instance 2, should be user 1
```

```
print("Meeting instance 2 participants: ", program.show_current_participants(2))
# Show participants of meeting instance 3, which doesn't exist
print(program.show_current_participants(3))

Meeting instance 1 participants:  []
User 1 successfully joined meeting
Meeting instance 2 participants:  [1]
Invalid meeting instance ID
```

"Show active meetings".The `show_active_meetings` is responsible for displaying the active meetings. It performs the following steps:

- It retrieves the list of all meetings from the Redis database.
- It filters the list to only include meetings that are currently active.
- Finally, it returns the list of active meetings.

```python
def show_active_meetings():
    # select the main database
    redis_db.select(0)
    active_meetings = []
    meeting_instances = redis_db.keys("meeting_instance:*")
    for instance_key in meeting_instances:
        # instance = redis_db.hgetall(instance_key)
        instance = redis_db.hmget(instance_key, ["isActive", "meetingID"])
        if instance[0] == "True":
            meeting_id = int(instance[1])
            active_meetings.append(meeting_id)

    return active_meetings
```

Listing 5: Show active meetings function

Here's a demonstration of the `show_current_participants` function:

```
# Show active meetings, should be meeting instance 2
print("List of active meetings: ", program.show_active_meetings())

List of active meetings:  [2]
```

" When a meeting ends, all participants must leave".The `end_meeting` is responsible for ending a meeting. It performs the following steps:

- It first checks if the provided `meeting_instance_id` is valid by retrieving the meeting instance information from the Redis database. If the meeting instance doesn't exist, it returns an "Invalid meeting instance ID" message..
- It checks if the meeting instance is currently active. If it's not active, it returns a message stating that the meeting is already ended.
- It calls `leave_meeting` for every participant in the meeting to ensure that all participants are removed from the meeting.
- It updates the meeting instance status in the Redis database to indicate that it's no longer active.
- It logs a "timeout" event in the Redis database.
- Finally, it returns a success message indicating that the meeting has been successfully ended.

```python
def end_meeting(meeting_instance_id):
    # select the main database
    redis_db.select(0)

    # check if meeting instance exists
    meeting_instance = redis_db.hgetall(f"meeting_instance:{meeting_instance_id}")
    if not meeting_instance:
        return "Invalid meeting instance ID"

    # check if meeting instance is active
    if meeting_instance["isActive"] == "False":
        return "Meeting instance is not active"

    participants = show_current_participants(meeting_instance_id)
    # event for each participant leaving the meeting
    for participant in participants:
        leave_meeting(participant, meeting_instance_id)

    # Update event log
    redis_db.select(1)

    event_id = redis_db.incr("event_id")
    event = {
        "eventID": event_id,
        "meetingInstanceID": meeting_instance_id,
        "event_type": "timeout",
        "timestamp": datetime.now().isoformat(),
    }
    redis_db.hmset(f"event:{event_id}", event)
    redis_db.select(0)

    # Deactivate the meeting instance
    redis_db.hset(f"meeting_instance:{meeting_instance_id}", "isActive", "False")

    return f"Meeting instance {meeting_instance_id} ended"
```

Listing 6: End meeting function

Here's a demonstration of the `end_meeting function` function:

```
# End meeting instance 2
print(program.end_meeting(2))
# End meeting instance 1, which is not active
print(program.end_meeting(1))
 # Try to end a non-existent meeting instance
print(program.end_meeting(3))
```

```
Meeting instance 2 ended
Meeting instance is not active
Invalid meeting instance ID
```

"A user posts a chat message".The `post_chat_message` is responsible for allowing users to post chat messages in a meeting. It performs the following steps:

- It first retrieves the meeting instance information based on the provided `meeting_instance_id`. If the meeting instance doesn't exist, it returns an "Invalid meeting instance ID" message.
- It checks if the user has already joined the meeting by calling the `check_user_in_meeting` function. If the user is not in the meeting, it returns a message stating that the user is not in the meeting.
- It checks if the meeting instance is active. If it's not active, it returns a message stating that the meeting is not currently active.
- It generates a new message ID and creates a new message entry in the chat log in the Redis database. The message entry includes the user ID, meeting instance ID, message content, and timestamp.
- Finally, it returns a success message indicating that the chat message has been successfully posted.

```python
def post_chat_message(user_id, meeting_instance_id, message):
    # select the main database
    redis_db.select(0)

    # check if meeting instance exists
    meeting_instance = redis_db.hgetall(f"meeting_instance:{meeting_instance_id}")
    if not meeting_instance:
        return "Invalid meeting instance ID"

    # check if user has joined the meeting
    user_joined = check_user_in_meeting(user_id, meeting_instance_id)
    if not user_joined:
        return "User is not in the meeting"

    # check if meeting instance is active
    if meeting_instance["isActive"] == "False":
        return "Meeting instance is not active"

    message_id = redis_db.incr("message_id")
    chat_message = {
        "messageID": message_id,
        "userID": user_id,
        "meetingInstanceID": meeting_instance_id,
        "message": message,
        "timestamp": datetime.now().isoformat(),
    }
    redis_db.select(0)
    redis_db.hmset(f"chat_message:{message_id}", chat_message)

    return "Chat message posted"
```

Listing 7: Post message function

Here's a demonstration of the `post_chat_message` function:

```
# User 1 posts a chat message in meeting instance 2, which they have not joined
print(program.post_chat_message(1, 2, "Hello, everyone!"))
# User 1 joins meeting instance 2
print(program.join_meeting(1, 2))
# User 1 posts a chat message in meeting instance 2
print(program.post_chat_message(1, 2, "Hello, everyone!"))
```

```
# User 2 posts a chat message in meeting instance 1, which is not active
print(program.post_chat_message(2, 1, "Hello world!"))
# Try to post a chat message with an invalid meeting instance ID
print(program.post_chat_message(1, 3, "Invalid meeting"))
```

```
User is not in the meeting
User 1 successfully joined meeting
Chat message posted
User is not in the meeting
Invalid meeting instance ID
```

”Show meeting’s chat messages in chronological order”.The `get_chat_messages` is responsible for retrieving the chat messages posted in a meeting. It performs the following steps:

- It first checks if the provided `meeting_instance_id` is valid by retrieving the meeting instance information from the Redis database. If the meeting instance doesn’t exist, it returns an ”Invalid meeting instance ID” message.
- It retrieves the chat messages associated with the provided meeting instance ID from the chat log in the Redis database.
- If there are chat messages available, it returns a list of the messages.
- If there are no chat messages available, it returns an empty array.

```python
def get_chat_messages(meeting_instance_id):
    # select the main database
    redis_db.select(0)

    # check if meeting instance exists
    meeting_instance = redis_db.hgetall(f"meeting_instance:{meeting_instance_id}")
    if not meeting_instance:
        return "Invalid meeting instance ID"

    chat_messages = []
    messages = redis_db.keys(f"chat_message:*")

    for message_key in messages:
        message = redis_db.hgetall(message_key)
        if message["meetingInstanceID"] == str(meeting_instance_id):
            chat_messages.append(message)

    chat_messages.sort(key=lambda x: x["timestamp"])
    return chat_messages
```

Listing 8: Messages in order function

Here’s a demonstration of the `get_chat_messages` function:

```
print(program.get_chat_messages(1)) # Retrieve chat messages for meeting instance 1
print(program.get_chat_messages(2)) # Retrieve chat messages for meeting instance 2
print(program.get_chat_messages(3)) # Try to retrieve chat
messages for an invalid meeting instance
```

```
[{'messageID': '2', 'userID': '2', 'meetingInstanceID': '1',
'message': 'Hi, John!', 'timestamp': '2023-05-08T10:12:00'}]
[{'messageID': '1', 'userID': '1', 'meetingInstanceID': '2',
'message': 'Hello, everyone!', 'timestamp': '2023-05-10T23:47:42.639477'}]
Invalid meeting instance ID
```

"Show for each active meeting when (timestamp) current participants joined ". The `get_participant_join_time` is responsible for retrieving the join time of participants in a meeting. It performs the following steps:

- It first checks if the provided `meeting_instance_id` is valid by retrieving the meeting instance information from the Redis database. If the meeting instance doesn't exist, it returns an "Invalid meeting instance ID" message.
- It checks if the meeting instance is active. If it's not active, it returns a message stating that the meeting is not currently active.
- It retrieves the join time of all participants associated with the provided meeting instance ID from the event log in the Redis database.
- If participants' join times are available, it returns a dictionary containing the user IDs as keys and their respective join times as values.
- If no participants' join times are available (e.g., no participants joined the meeting), it returns an empty dictionary.

```python
def get_participant_join_time(meeting_instance_id):
    # select the main database
    redis_db.select(0)

    # check if meeting instance exists
    meeting_instance = redis_db.hgetall(f"meeting_instance:{meeting_instance_id}")
    if not meeting_instance:
        return "Invalid meeting instance ID"

    # check if meeting instance is active
    if meeting_instance["isActive"] == "False":
        return "Meeting instance is not active"

    participants = show_current_participants(meeting_instance_id)

    # select the event database
    redis_db.select(1)

    participants_join_time = {}

    for participant in participants:
        events = redis_db.keys(
            f"event:*:userID:{participant}:meetingInstanceID:{meeting_instance_id}:
    event_type:join_meeting"
        )
        join_events = []
        for event_key in events:
            event = redis_db.hgetall(event_key)
            join_events.append(event)

        join_events.sort(key=lambda x: x["timestamp"])
        participants_join_time[participant] = join_events[-1]["timestamp"]

    return participants_join_time
```

Listing 9: Get join time function

Here's a demonstration of the `get_participant_join_time` function:

```
# Retrieve join times of all participants in meeting instance 2,
which only user 1 has joined
print(program.get_participant_join_time(2))
```

```
{1: '2023-05-10T23:47:42.636386'}
```

” **Show for an active meeting and a user his/her chat messages**.The `get_user_messages`is responsible for retrieving all the chat messages sent by a specific user in a particular meeting instance. It performs the following steps:

- It first checks if the provided `meeting_instance_id` is valid by retrieving the meeting instance information from the Redis database. If the meeting instance doesn't exist, it returns an ”Invalid meeting instance ID” message.
- It retrieves all the chat messages associated with the provided `user_id` and `meeting_instance_id` from the chat log in the Redis database.
- If chat messages are available for the user in the specified meeting instance, it returns a list containing all the messages, sorted by timestamp.
- If no chat messages are available (e.g., the user hasn't sent any messages in the specified meeting instance), it returns an empty list.

```python
def get_user_messages(meeting_instance_id, user_id):
    # select the main database
    redis_db.select(0)

    # check if meeting instance exists
    meeting_instance = redis_db.hgetall(f"meeting_instance:{meeting_instance_id}")
    if not meeting_instance:
        return "Invalid meeting instance ID"

    messages = []

    message_keys = redis_db.keys(f"chat_message:*")

    for message_key in message_keys:
        message = redis_db.hgetall(message_key)
        if message["meetingInstanceID"] == str(meeting_instance_id) and message[
            "userID"
        ] == str(user_id):
            messages.append(message)

    messages.sort(key=lambda x: x["timestamp"])

    return messages
```

Listing 10: Get user messages function

Here's a demonstration of the `get_user_messages` function:

```
 # Retrieve all chat messages sent by user 1 in meeting instance 2
print(program.get_user_messages(2, 1))
```

```
[{'messageID': '1', 'userID': '1', 'meetingInstanceID': '2',
'message': 'Hello, everyone!', 'timestamp': '2023-05-10T23:47:42.639477'}]
```

Events demonstration

It is worth demonstrating that `join_meeting` and `leave_meeting` and timeout events are logged in the event log.

```
# Reset redis
!python redis_init.py

print(program.join_meeting(1, 2)) # Logs event
print(program.post_chat_message(1, 2, "Hello, everyone!")) # Doesn't log event
print(program.leave_meeting(1, 2)) # Logs event
print(program.end_meeting(2)) # Logs event
```

```
Redis initialized with sample data.
User 1 successfully joined meeting
Chat message posted
User successfully left the meeting
Meeting instance 2 ended
```

The events can be seen at the bottom of the following redis snapshot:

```
!python printRedis.py
```

```
    Users:
User ID: 1
    userID: 1
    name: John Doe
    age: 25
    gender: Male
    email: john.doe@example.com

User ID: 2
    userID: 2
    name: Jane Smith
    age: 30
    gender: Female
    email: jane.smith@example.com

Meetings:
Meeting ID: 1
    meetingID: 1
    title: Team Standup
    description: Daily standup meeting for the team
    isPublic: True
    audience: []

Meeting ID: 2
    meetingID: 2
...
    meetingInstanceID: 2
```

```
event_type: timeout
timestamp: 2023-05-10T23:47:43.102811
```