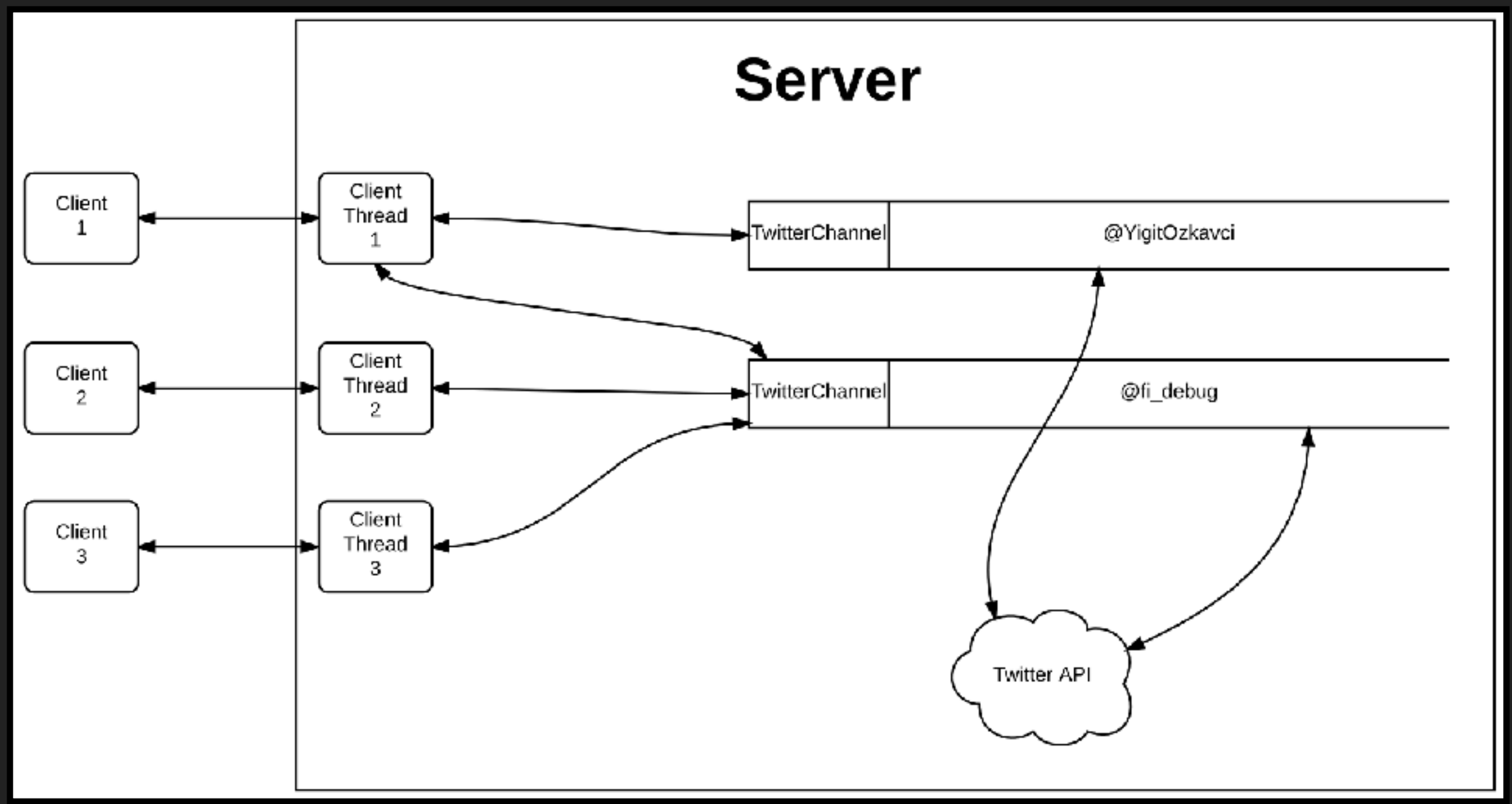
A DISTRIBUTED TWITTER WATCHER

FOLLOW-IT

GENERAL IDEA



CLIENT THREADS

- Each thread is responsible of communicating with one and only one client.
- They parse the incoming messages from clients into “Command”s, and let commands take the necessary actions.
- The only objects that are allowed to send & receive messages to & from threads..

COMMANDS

- The object that knows what to do with a valid client message.
- Can call methods on the client by which message is received.
- Can decide whether the client is authorized to take the action or not (e.g. unregistered clients cannot subscribe).

TWITTER AGENTS

- Watchers of tweets. Focused on just one Twitter user, they fetch new tweets every 5 seconds, and notify the watchers of that user.
- They are spawned every time a unique Twitter user is wanted to be watched by a client.
- They are essentially TimerTasks, but specialized on Twitter users.

BROADCASTING

- Whenever an update on a Twitter account is detected by the watcher of that user, this update is broadcasted to all users

- For every user, we are making request to Twitter API every 5 seconds. No duplicate requests for the same user is being made

- Twitter API allows us to send 900 requests per 15 minute, so we are limited to 5 users being watched concurrently. Any number of clients can subscribe to these users though

MESSAGING PROTOCOL

- A message format is enforced for the server-client communication.
- Every message should have a valid JSON format, or these messages are rejected
- Every message should have a top-level “tag” key, along with a valid message tag

MESSAGE TAGS (CLIENT -> SERVER)

- REGISTER: Client wants to register. “username” should be provided inside data key.
  Example:
  { “tag”: “REGISTER”, “data”: { “username”: “Yigit Ozkavci” } }

- SUBSCRIBE: Client wants to subscribe. “channel” and “user” should be provided inside data key.
  Example:
  { “tag”: “SUBSCRIBE”, “data”: { “channel”: “Twitter”, “user”: “YigitOzkavci” } }

MESSAGE TAGS (SERVER -> CLIENT)

- REGISTER_ACCEPT: Server accepted client’s registration request. Client can perform subscription action from now on.
  Example:
  { “tag”: “REGISTER_ACCEPT” }

- SUBSCRIPTION_ACCEPT: Server accepted client’s subscription request. Server will start to send updates about the watched account to client from now on.
  Example:
  { “tag”: “SUBSCRIPTION_ACCEPT”, “data”: { “channel”: “Twitter” } }

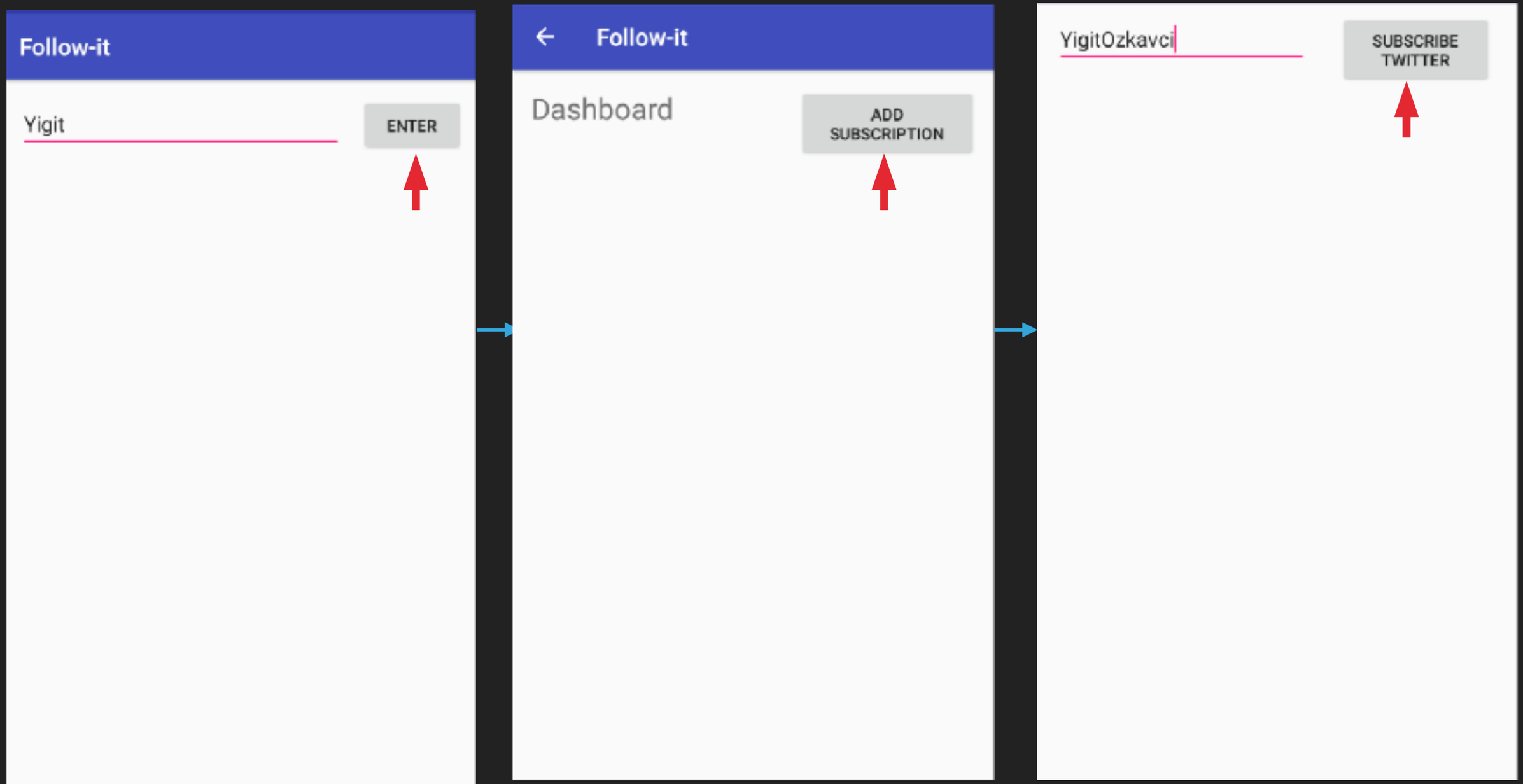
- SUBSCRIPTION_REJECT: Server rejected client’s subscription request. There won’t be any difference in server state.
  Example:
  { “tag”: “SUBSCRIPTION_REJECT”, “data”: { “channel”: “Twitter” } }

MESSAGE TAGS (SERVER -> CLIENT, CONTINUED)

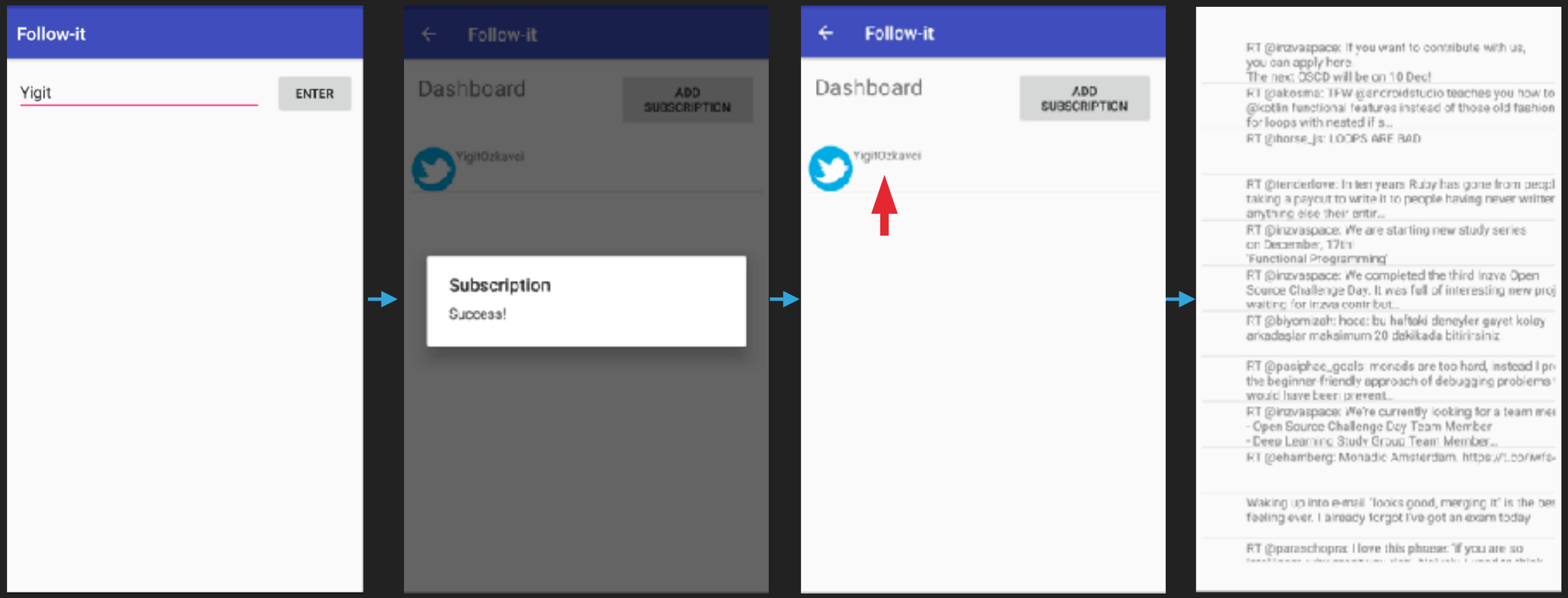
- ERROR: There was an error either caused by the server, or client. This tag includes protocol errors (e.g. invalid message format), as well as subscription errors (e.g. Twitter user cannot be found)
  Example:
  { “tag”: “ERROR”, “data”: { “message”: “Twitter user YigitOzkavci cannot be found” } }

- TWEETS: Server is sending tweet data to client. This message should contain which user this update is for, as well as the actual data.
  Example:
  { “tag”: “TWEETS”, “data”: { “tweets”: [...], “user”: “YigitOzkavci” } }

USER INTERFACE



USER INTERFACE



STORAGE & DATA MODEL

- Every data on Android application is persistent. Meaning they won't disappear if we restart the whole application.
- There are two database models:
  - Subscription(id, channel, username): Represents a single subscription. A client may have multiple subscriptions.
  - SubscriptionData(id, subscription_id, data): Represents a single data coming from subscription with id `subscription_id`. In practice, this data is just a string representing a single tweet.
- All database access is, and must be asynchronous; and it's done via AsyncTasks.

COMMUNICATION VIA SOCKET

- A FIConnection instance is managing the whole socket communication cycle.
- FIConnection exposes an API to application for abstracting away the socket message details. For instance, to subscribe to channel for a user, one should call the following function:

  subscribe(Channel channel, String user, TaskListener listener);
- Registration and subscription are messages that cause a response as answer. “Waiting for an answer” is a bad practice, so whenever one should call register() or subscribe(), s/he should also provide a listener as callback.
- All socket message sendings are done in “fire and forget” manner. According to response, async listener modifies the FIConnection state and/or writes necessary data to database.

THANK YOU.

YİĞİT ÖZKAVCI