Department of Information Engineering, CUHK
MScIE – 2nd Semester, 2015/16

IEMS 5722
Mobile Network Programming and Distributed Server Architecture

Lecture 11
Web Sockets for Real-time Communications

Lecturer: Albert C. M. Au Yeung

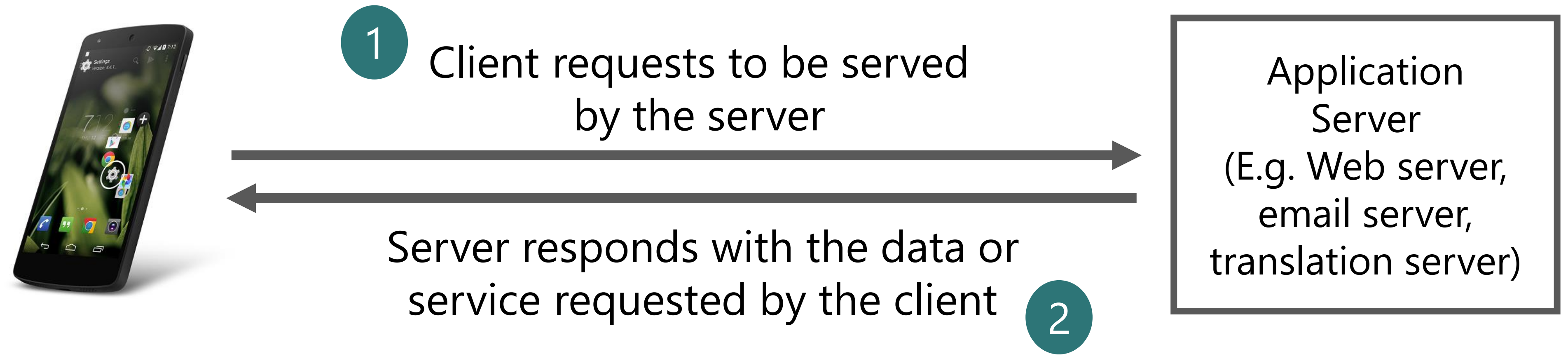
24th March, 2016

Web Sockets

Limitations of HTTP

All of the examples we have gone through in network programming so far can be regarded as using the "**pull**" method

- Communication is always initiated by the client
- Client "pulls" data or services from the server when necessary (e.g. when the user launches the app, or presses a button)


1
Client requests to be served by the server
Server responds with the data or service requested by the client
2
Application Server (E.g. Web server, email server, translation server)

Limitation of HTTP

HTTP is a pull-based protocol

- Users browse the Web and actively decide which Website to browse, which link to follow
- A effective and economical way (every user chooses what he needs)
- However, if some resources are regularly requested, the pull model can put heavy load on the server

Implementing Push

The World Wide Web, and in particular the HTTP protocol, is designed for "pull", and additional engineering is required to implement push on the Web'

Some ways to "emulate" push on the Web

- Polling (Periodic pull)
- The Comet model
- BOSH
- WebSockets

WebSocket

- A protocol providing full-duplex communications channels between two computers over a TCP connection
- Designed to be implemented in Web browsers and Web servers
- Communications are done over TCP port 80 (can be used in secured computing environments)
- Supported in Chrome 14, Safari 6, Firefox 6, IE 10 or later versions

HTTP is half-duplex: only one side can send data at a time, like walkie-talkie

Reference:
http://tools.ietf.org/html/rfc6455
http://www.websocket.org/

WebSocket

- A persistent connections between a Web browser (or a mobile app) and a server
- Both sides can send out data to the other side at any time
- Why WebSocket?
  - Lower latency (avoid TCP handshaking)
  - Smaller overhead (only 2 bytes per message)
  - Less unnecessary communication (data is only sent whenever needed)

WebSocket

WebSocket is part of the HTML5 standard

- Supported in latest versions of major Web browsers
- Simple API in JavaScript
- Libraries also available on iOS and Android

```javascript
var host = 'ws://localhost:8000/example';
var socket = new WebSocket(host);

socket.onopen = function() {
    console.log('socket opened');
    socket.send('Hello server!');
}
socket.onmessage = function(msg) {
    console.log(msg);
}
socket.onclose = function() {
    console.log('socket closed');
}
```

Try the game at
http://chrome.com/supersyncsports/

WebSocket

Particularly useful when you would like to develop applications such as:

- Real-time multiplayer games
- Chat rooms
- Real-time news feed
- Collaborative apps (e.g. consider something like Google Documents)
- Live commenting
- …

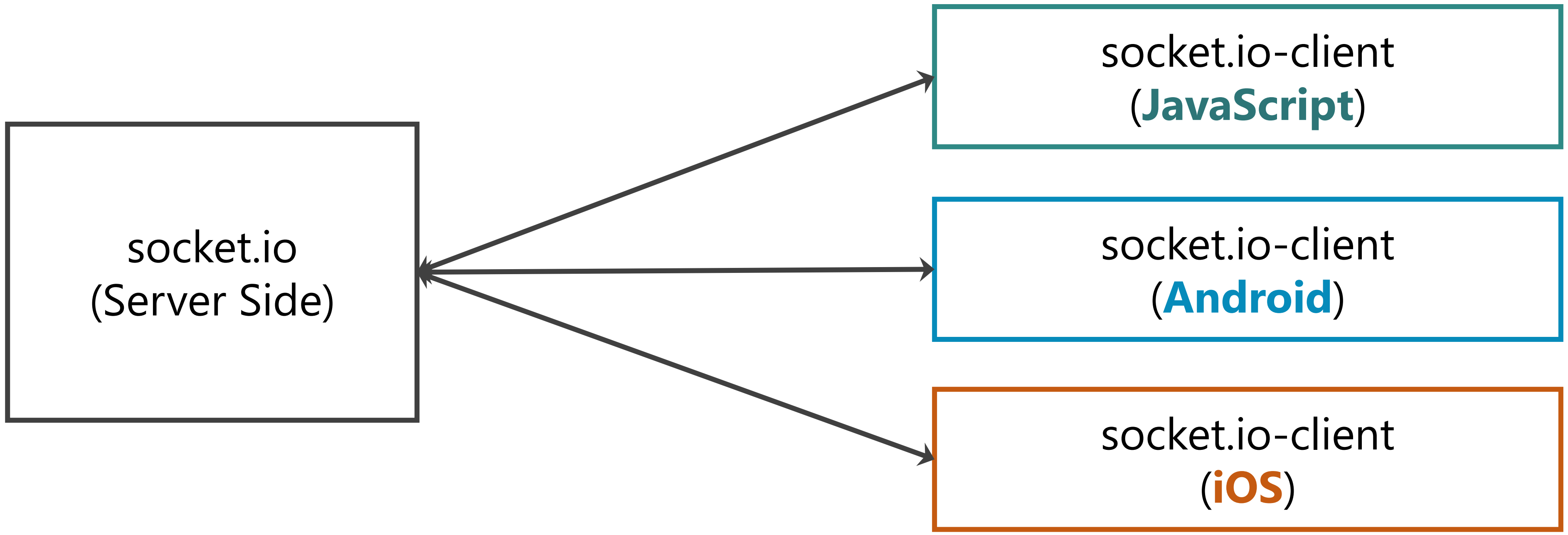
socket.io

socket.io

- A library based on **node.js** for real time, bi-directional communication between a server and one or multiple clients
- Using WebSocket to perform data communication (fall back to older solutions when WebSocket is not supported)
- Official Website: http://socket.io/
- Originally written for node.js on the sever side and JavaScript on the client side, there are now libraries for Python, Android and iOS

socket.io

- socket.io has two parts: 1) Server and 2) Client
- Client libraries are available in JavaScript (Web), Android and iOS
- Allow you to build real-time apps across multiple platforms



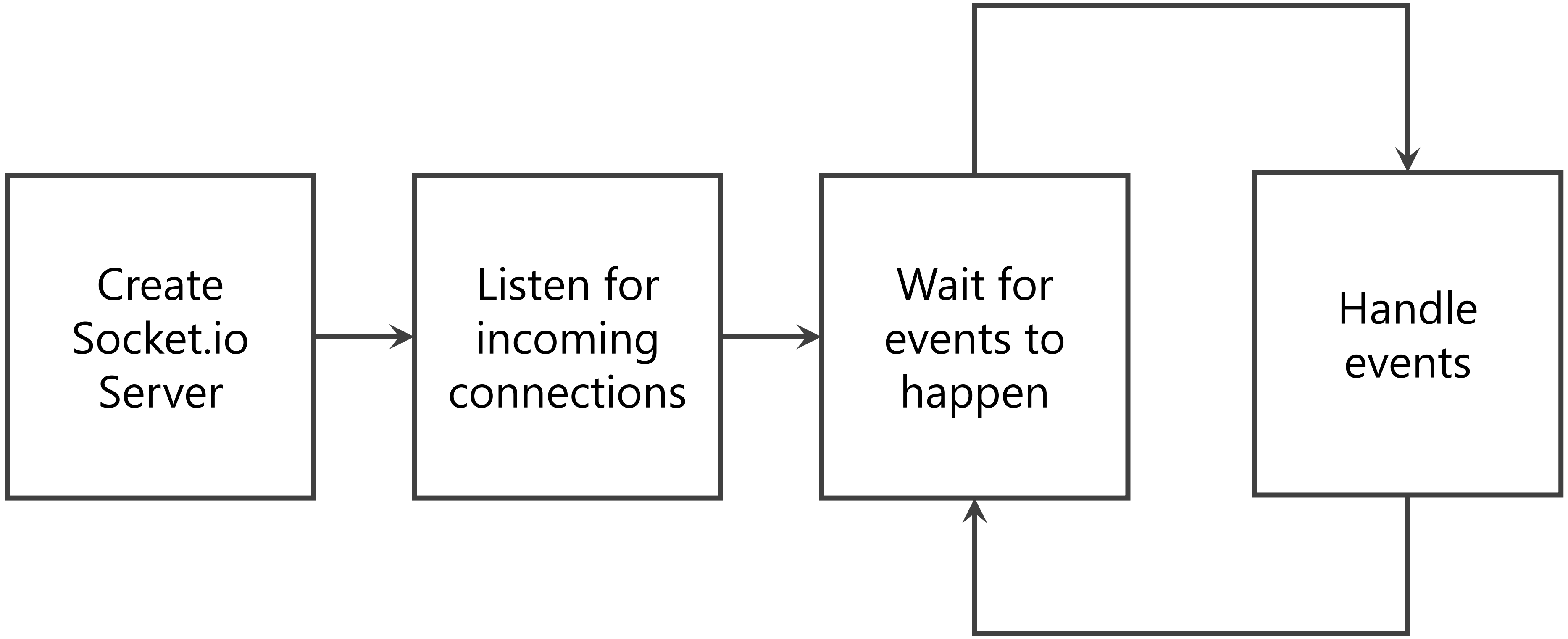
References
iOS Client: http://socket.io/blog/socket-io-on-ios/
Android: http://socket.io/blog/native-socket-io-and-android/

socket.io

- **Event-driven**
  - ➢ Once connected, the server and client can communicate with each other by triggering or "emitting" events
  - ➢ Create callback functions to carry out different actions whenever some events happens

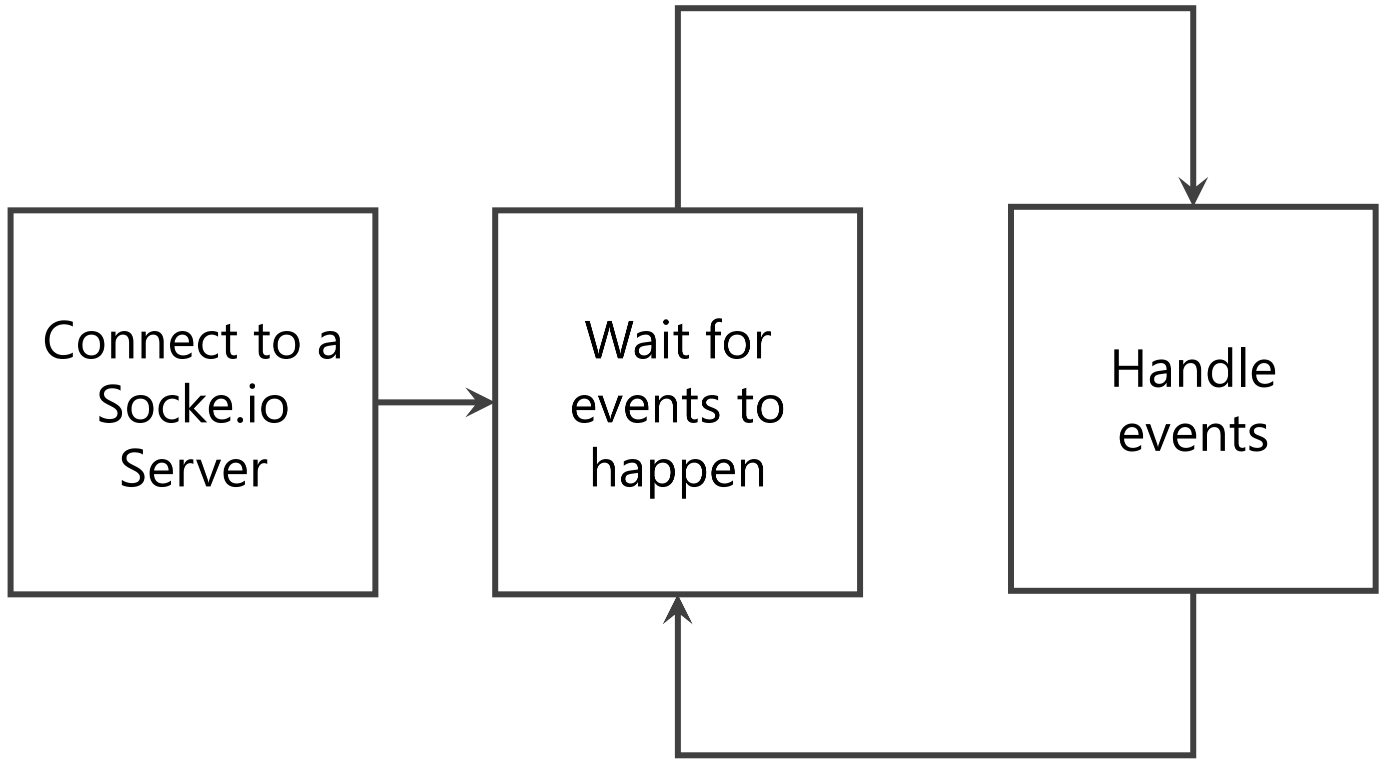
socket.io

- Flow of creating a server-side program



socket.io

- Flow of creating a client-side program



Flask-SocketIO

Flask-SocketIO

- A module that allows you to use socket.io in Flask applications
- http://flask-socketio.readthedocs.org/
- Install using the following command:

```
$ pip install flask-socketio
```

- Note: install the concurrent networking library Eventlet as well:

```
$ pip install eventlet
```

Flask-SocketIO

- Initialization

```
from flask import Flask, render_template
from flask_socketio import SocketIO

app = Flask(__name__)
app.config['SECRET_KEY'] = ‘secretkey’

socketio = SocketIO(app)

if __name__ == ‘__main__’:
    socketio.run(app)
```

You need to provide a secret key for Flask to encrypt data for user sessions

Setup the app to use functions of socket.io

This is for debugging purpose, we will discuss how to deploy applications that using socket.io soon

Flask-SocketIO

- Remember we mentioned that in Socket.io all communications are based on events
- We will have to define handlers of events in our Flask application
- For example:

```
@socketio.on('message')
def handle_message(message):
    # Actions to be performed when a message is received
    # ...
```

The name of the event (NOTE: some names cannot be used because they already represent some special events)

Events

- Both the server and the client can generate events, and if the other side has a handler of that event, the handler will be invoked to carry out some actions
- In Flask-SocketIO, there are several different types of evetns
  - Special events ('connect', 'disconnect', 'join', 'leave')
  - Unnamed events ('message' or 'json')
  - Custom events (a name of your choice, e.g. 'my event')

Events

- Connection events

```
# Will be invoked whenever a client is connected
@socketio.on('connect')
def connect_handler():
    print "Client connected. “

# Will be invoked whenever a client is disconnected
@socketio.on('disconnect')
def disconnect_handler():
    print "Client disconnected."
```

Events

- Unnamed events

```
# Will be invoked whenever an unnamed event happens
@socketio.on('message')
def message_handler(message):
    print "message received: %s" % message

# Will be invoked whenever an unnamed event happens (with JSON data)
@socketio.on('json')
def disconnect_handler(json):
    print "json received: %s" % str(json)
```

Events

- Custom events are events with custom-defined names

```
# Will be invoked when the client emits an event of the type “event_001”
@socketio.on('event001')
def event001_handler(json):
    print "json received: %s" % str(json)
```

Events

- The server can send message to clients by creating events using the **`send()`** function or the **`emit()`** function
- The **`send()`** function is for sending unnamed events
- The **`emit()`** function is for sending custom named events

```python
from flask_socketio import send, emit

@socketio.on('message')
def handle_message(message):
    send(message)

@socketio.on('json')
def handle_json(json):
    send(json, json=True)

@socketio.on('event001')
def handle_my_custom_event(json):
    emit('event001', json)
```

Broadcasting

- Normally, **send** and **emit** only send message to
- Broadcasting allows you to send a message to all clients who are connected to the server
- For example, in a multi-player game, one user performs an action, and you want this to be known to all other users

```
@socketio.on('event001')
def handle_event001(data):
    emit('event001', data, broadcast=True)
```

Set broadcast=True when emitting a message

Rooms

- In some applications, users may interact with only a subset of other users
- Examples:
  - Chat application with multiple rooms
  - Multiplayer games (multiple game boards, game rooms, etc.)
  - …

```
from flask_socketio import join_room, leave_room

@socketio.on('join')
def on_join(data):
    username = data['username']
    room = data['room']
    join_room(room)
    send('user_join', '%s joined' % username,  room=room)

@socketio.on('leave')
def on_leave(data):
    username = data['username']
    room = data['room']
    leave_room(room)
    send('user_leave', '%s left' % username, room=room)
```

Deploying Flask-SocketIO

- If you have "eventlet" installed, you can use the embedded server which is production-ready
- Invoked by `socketio.run(app, host="0.0.0.0", port=5000)`
- Sample supervisor configuration files:

```
[program:iems5722_socket]
command = python app.py
directory = /home/albert/iems5722
user = albert
autostart = true
autorestart = true
stdout_logfile = /home/albert/iems5722/app.log
redirect_stderr = true
```

Deploying Flask-SocketIO

- If you would like to deploy a single Flask app including both your APIs and the socket.io event handlers, use Gunicorn.
- For example:

```
[program:iems5722]
command = gunicorn --worker-class eventlet -w 1 app:app –b localhost:8000
directory = /home/albert/iems5722
user = albert
autostart = true
autorestart = true
stdout_logfile = /home/albert/iems5722/app.log
redirect_stderr = true
```

Socket.io on Android

socket.io on Android

- A socket.io client library for Java and Android
- Source code and set-up guide: https://github.com/socketio/socket.io-client-java
- To use the library in your Android project, add the following dependency to your **`build.gradle`** file under the app directory

```
compile ('io.socket:socket.io-client:0.7.0') {
    exclude group: 'org.json', module: 'json '
}
```

socket.io on Android

- Creating a socket.io client in Android

```
private Socket socket;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.my_activity);

    try {
        socket = IO.socket("http://my.server.com/");
    } catch (URISyntaxException e) {
        throw new RuntimeException(e);
    }
    socket.on(Socket.EVENT_CONNECT_ERROR, onConnectError);
    socket.on(Socket.EVENT_CONNECT_TIMEOUT, onConnectError);
    socket.on(Socket.EVENT_CONNECT, onConnectSuccess);
    socket.on("event001", onEvent001);
    socket.connect();
}
```

The URL of the socket.io server

Define callback functions (handlers) of different events

Initiating a connection to the server

socket.io on Android

- Creating event handlers/listeners:

```
private Emitter.Listener onConnectSuccess = new Emitter.Listener() {

    @Override
    public void call(Object... args) {
        JSONObject data = (JSONObject) args[0];
        ...
        ...
    }
};
```

You must override this function "call" in your listener

This function is called on a new thread, **NOT** the UI Thread. You should **NOT** manipulate UI components here.

There can be multiple parameters passed to this function (depending on the server). If it is JSON, cast it to a JSONObject or JSONArray.

socket.io on Android

- Sending string message back to the server

```
socket.emit(“event001”, “data”);
```

- You can also send JSON objects in the emit function:

```
JSONObject json = new JSONObject();
json.put(“name”, “Peter”);
json.put(“gender”, “male”);
socket.emit(“event001”, json);
```

socket.io on Android

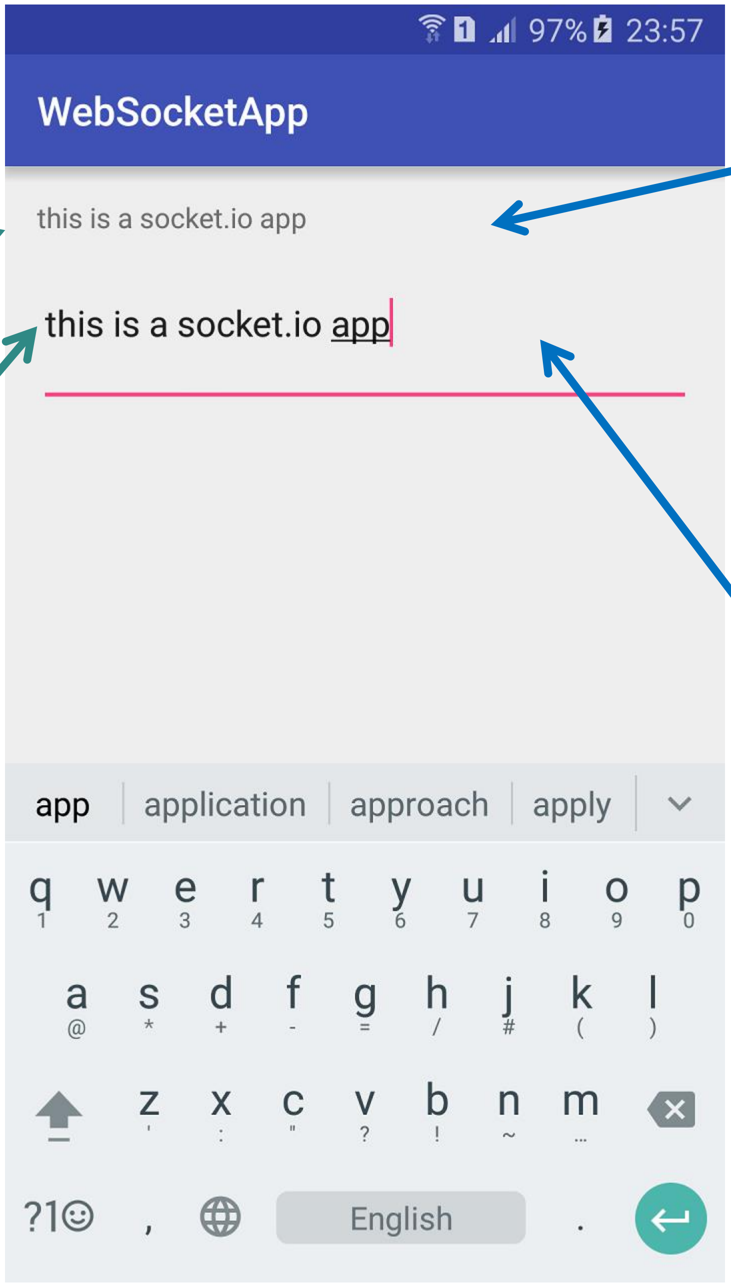
- Like many other things on Android, you need to manage the socket's life cycle: when the user leaves the activity, you should disconnect

```
@Override
public void onDestroy() {
    super.onDestroy();
    socket.disconnect();
    socket.off();
}
```

Use the "off" method to remove the event listeners

Example:
Real-time Typing Update

Example



this is a socket.io app

this is a socket.io app

A TextView for displaying what the server has sent out

An EditText for user to input text

Client receives the event, and update the TextView with what is received from the server

Server receives the message, and emit another event to all connected clients

When the text in this input area is changed, a event is emitted to the server

Example – Server Application

```
from flask import Flask, render_template
from flask_socketio import SocketIO, emit

app = Flask(__name__)
app.config['SECRET_KEY'] = 'iems5722 ‘
socketio = SocketIO(app)

@socketio.on('text')
def update_handler(json):
    emit( 'update', {'text': json['text']}, broadcast=True)
    return

if __name__ == '__main__':
    socketio.run(app, host="0.0.0.0", port=5000)
```

Example – Android App

- Source code available at: https://gist.github.com/albertauyeung/884137ed0e39944d63cd

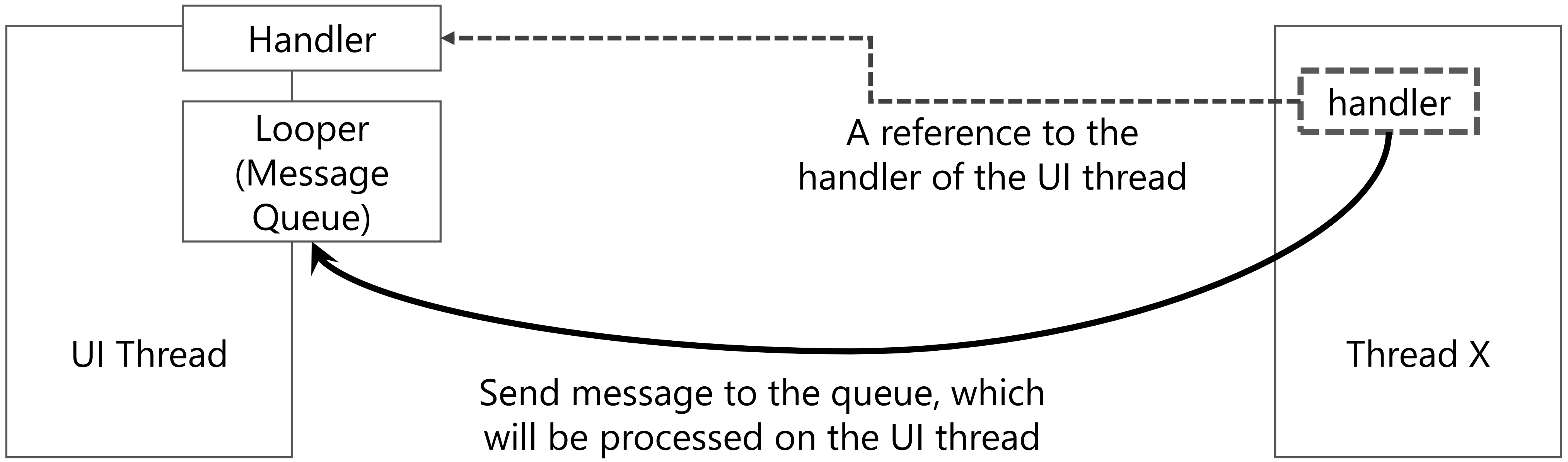
Multithreading and Handlers

Handlers

Handlers is a component in the Android system for managing threads.

A **Handler** object:

- Is associated with a particular thread and its message queue called "looper"
- Receives messages and runs code to handle the message


Handler
Looper (Message Queue)
UI Thread
handler
Thread X
A reference to the handler of the UI thread
Send message to the queue, which will be processed on the UI thread

Handlers

- A handler will be associated with the thread in which it is created
- You need to override the **`handleMessage()`** method to specify the action(s) to be taken when a message is received

```
public class MyActivity extends Activity {

    private static class MainHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            ...
        }
    }

    Handler handler = new MainHandler();

    ...
```

Manipulation of UI components can be done here, as this will be performed on the UI thread

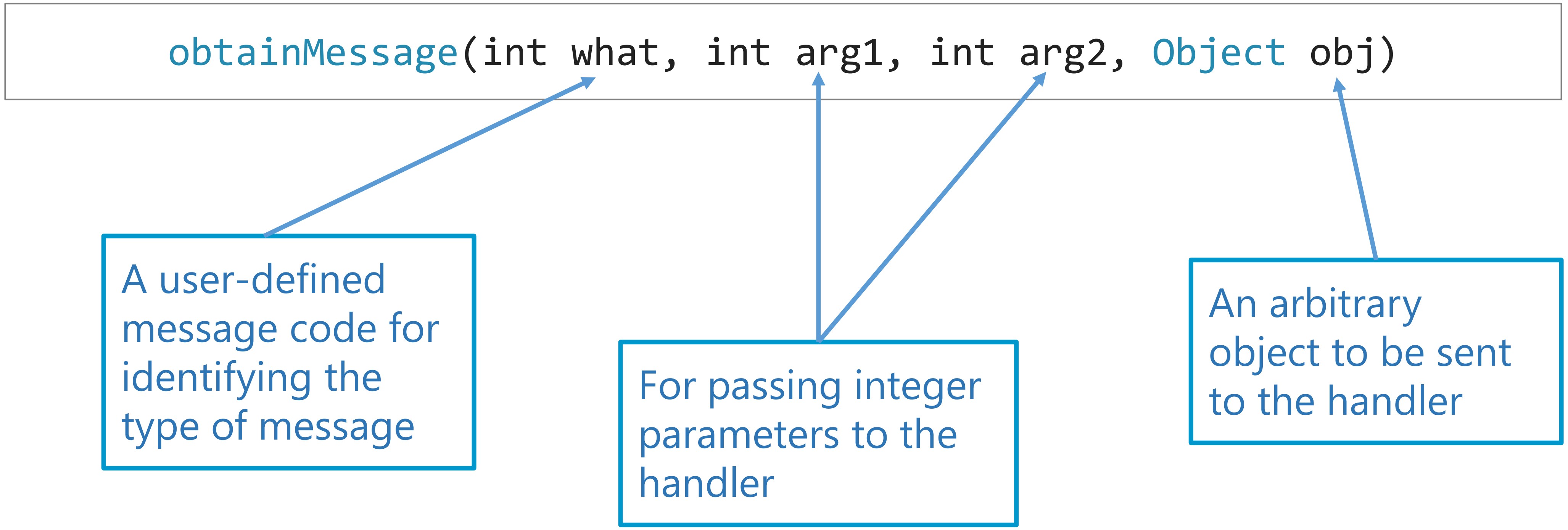
Handlers

- When you want to ask the UI thread to perform something, send a message to its handler

```
...
Runnable runnable = new Runnable() {
    public void run() {
        ...
        String data = “Message to send.”;
        Message msg = handler.obtainMessage(ACTION_CODE, data);
        msg.sendToTarget();
    }
};
...
```

Handlers

- A message can be created by using the `Handler.obtainMessage(...)` method
- You can supply up to four parameters to this method



```
obtainMessage(int what, int arg1, int arg2, Object obj)
```

A user-defined message code for identifying the type of message

For passing integer parameters to the handler

An arbitrary object to be sent to the handler

Reference: https://developer.android.com/reference/android/os/Handler.html

Handlers

- To extract parameters from the message in the `handleMessage()` method:

```
...
mHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        if (msg.what == ACTION_CODE_1) {
            int arg1 = msg.arg1;
            int arg2 = msg.arg2;
            MyObject = (MyObject)msg.obj;
            ...
        } else if (msg.what == ACTION_CODE_2) {
            ...
        }
        ...
    }
};
```

Reference: https://developer.android.com/reference/android/os/Message.html

Embedding Data in Messages

- You can also embed data in the message using the **setData()** method

```
...
Message msg = new Message();
Bundle data = new Bundle();
data.putString(“PARAM_1”, “String...”);
data.putInt(“PARAM_2”, 100);
...
msg.setData(data);
msg.what = ACTION_CODE_1;
handler.sendMessage(msg);
```

- Then in the **handleMessage()** method you can retrieve the data like this:

```
String str = msg.getData().getString(“PARAM_1”);
```

Example – Android App

- Using handler and messages to handle communications between threads
  https://gist.github.com/albertauyeung/e38e13a8131a5b119c1c

End of Lecture 11