Application Development for Mobile and Ubiquitous Computing

5. Communication Mechanisms

Dr. Ing. Thomas Springer
Technische Universität Dresden
Chair of Computer Networks
Lecture Structure

Application Development

Mobile Business Applications

Cross-Platform Development

Mobile Web Applications
Android
iOS

Mobile Middleware

Location-based Services
Disconnected Operations
Communication Mechanisms
Energy Awareness

Enabling Technologies and Challenges

Adaptation and Context-awareness

Dr. Thomas Springer
Application Development - 4. Disconnected Operations
Social Fitness App – Connectivity Challenge

- Uploading local workout data
  - Request/Response (e.g. HTTP)

![Cycling Image]

27.3 km: 26:39
"Erste Etappe war eine einzige Quälerei, beim nächsten Mal geht’s bestimmt besser!

Problems
- Heterogeneous access networks
- Varying quality and stability
RPC Principle

- Extension of local procedure call to remote call
- Goal: syntactic and semantic uniformity
  - call mechanism (transparent network communication)
  - language elements
  - error semantics
Synchronous communication mode requires established and stable network connection for RPC
- problem: frequent disconnections in wireless networks
- problem: high delay due to message loss when using wireless communication technologies
- problem: high energy consumption – user disconnects mobile device, application is blocked

Calls are sent according to logic program flow
- problem: in longer phases of disconnection no RPC calls can be sent and processed – application is blocked
- problem: no bundeling of calls to exploit temporarily available higher data rates

Binding between client and server at the beginning of the conversation
- problem: due to mobility and disconnections a rebinding to other servers is required but usually not supported
Mobile RPC Concept

- M-RPC enables
  - Reliable call mediation via unreliable connections
  - Optimized RPC communication
  - Dynamic (re-)binding
Mobile RPC Concept

- Proxy located at the base station within the network infrastructure
  - Adapted transport protocol between mobile device and base station
  - Requests stored in request cache until response is successfully received by client
  - Retransmission of requests performed by proxy
- Queuing of RPC calls and results at the mobile device and the proxy
- Cumulated calls -> bulking (throughput optimization)
Mobile RPC Concept

- Dynamic rebinding due to indirection via proxy
  - Client holds logical binding to server, physical binding at proxy
  - New physical binding can be established after disconnections or client movement
  - Server state has to be considered
Google Volley
Main Features

- **Limited Bandwidth**
  - Request prioritisation
  - Request cancellation

- **Connection and Transmission Errors**
  - Queuing
  - Result caching
  - Request resending
▲ Request
  - Represents HTTP-Request
  - Methods: GET, POST, PUT, DELETE, ...
  - Defined return type
    - Standard types usable (String, JSON)

▲ RequestQueue
  - Singleton, one instance per app
  - Responsible for scheduling and initiation of requests

▲ ResponseListener
  - Receives response or error
Google Volley
Request Initiation

Request

Listener

Listener

Queue

StringRequest req = new StringRequest(Request.Method.GET, "www.example.com",
new Response.Listener<String>() {
  @Override
  public void onResponse(String response) {
    // Do something with the response
  }
},
new Response.ErrorListener() {
  @Override
  public void onErrorResponse(VolleyError error) {
    // Handle error
  }
});

// Add the request to the queue
Volley.newRequestQueue(this).add(req);
• Timeout
• Number of retries
• Backoff time

Req.setRetryPolicy(
    new DefaultRetryPolicy(initialTimeoutMs, maxNumRetries, backoffMultiplier));

• For example

Req.setRetryPolicy(
    new DefaultRetryPolicy(1000, 3, 2.0f));
• **Transparent response cache**
• **Cache interface**

```java
Cache cache =
    new DiskBasedCache(getCacheDir(),
                        maxCacheSizeinBytes);
RequestQueue queue =
    new RequestQueue(cache, network);
```

```java
<<interface>>

    Cache
    get(key)
    put(key, value)
    remove(key)
    clear()

DiskBased

    Cache
    rootDir
    maxSize

    (default)

Own Cache Impl.
```
• Not out-of-the-Box
• User defined requests (derived from `StringRequest`):

```java
public abstract class PriorityRequest extends StringRequest {
    private Priority mPriority;

    public PriorityRequest(int method, String url,
                           Response.Listener<String> listener,
                           Response.ErrorListener errorListener,
                           Priority priority) {
        super(method, url, listener, errorListener);
        mPriority = priority;
    }

    @Override
    public Priority getPriority() {
        return mPriority;
    }
}
```

<table>
<thead>
<tr>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOW</td>
</tr>
<tr>
<td>NORMAL</td>
</tr>
<tr>
<td>HIGH</td>
</tr>
<tr>
<td>IMMEDIATE</td>
</tr>
</tbody>
</table>
• Distinct Cancellation-API
  • Individual requests
    ```java
    req.cancel();
    ```
  • Selection by tag
    ```java
    req.setTag("tag");
    ...
    queue.cancelAll("tag");
    ```
  • Selection by filter
    ```java
    queue.cancelAll(new LowPrioFilter());
    ```

```java
public class LowPrioFilter implements RequestQueue.RequestFilter {

    @Override
    public boolean apply(Request<?> req) {
        return (req.getPriority() == Request.Priority.LOW);
    }
}
```
Representational State Transfer - REST

- Model for distributed hypermedia systems
  - introduced by Roy Fielding (2000)

- Broadly adapted architecture style in the web
- Many systems provide a REST API
- Lightweight implementation compared to Web Services
  - Set of predefined operations
    - create, read, update, write (CRUD)
  - Multiple encoding formats
    -> e.g. compact encoding with JSON
Representational State Transfer - REST

- Based on a client/server architecture
- Uses **stateless** communication protocol
  - each request must contain all necessary information (no context stored on server), session states stored only on client
- In contrast to Remote Procedure Call (RPC) requests in a REST system are not directed to procedures but to **resources**
  - Resource: web page, data collection, image, ...
  - Every resource has to be available through a **unique identifier** (URI)
  - **HTTP methods** form set of operations (**Create**, **Read**, **Update**, **Delete**)
  - can have multiple representations (e.g. **XML** / **JSON**)

![Diagram showing the interaction between client and server with REST interface.]

1. GET document
2. 
3. Manipulate document
4. POST document
REST - URIs and Methods

**Resource timeline**

- http://myserver.com/timeline

**Resource media**

- Resource videoX

**Resource activities**

- Resource activity1
- Resource activity2

<table>
<thead>
<tr>
<th>Resource type</th>
<th>GET</th>
<th>POST</th>
<th>PUT</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>collection</strong></td>
<td>List URIs and details of resources in collection</td>
<td>Create or replace collection item with POST on parent</td>
<td>Create new entry in collection</td>
<td>Delete addressed collection</td>
</tr>
<tr>
<td></td>
<td><a href="http://myserver.com/timeline/activities">http://myserver.com/timeline/activities</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>item</strong></td>
<td>List details of addressed resource in appropriate format</td>
<td>Create or update addressed resource</td>
<td>Create new entry or update existing entry</td>
<td>Delete addressed resource</td>
</tr>
<tr>
<td></td>
<td><a href="http://myserver.com/timeline/activities/activity1">http://myserver.com/timeline/activities/activity1</a></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
JavaScript Object Notation

- Standard format specified as RFC 4627
- Programming language independent
- Text-based data encoding, Human readable
- JSON Syntax Rules (subset of the JavaScript object notation syntax)
  - Data is in name/value pairs
  - Data is separated by comma
  - Curly brackets hold objects
  - Square brackets hold arrays

Activity = {
  "user": "userURI",
  "type": "cycling",
  "distance": "120",
  "time": "05:21:12",
  "media": "videoXURI",
}

ActivityList = [
  { "user": "uri1", "type": "cycling" },
  { "user": "uri2", "type": "walking" },
  { "user": "uri3", "type": "running" }
]
REST - Implementation Principle

- Activity represents Screen in Android

- Service Layer
  - Application specific API, e.g. `getTimeline()`, `getActivity()`
  - Maps application-specific service calls to REST service calls
  - manages requests (e.g. request queue)
  - provides callback for UI activity

- REST Service (Service API)
  - network requests decoupled from UI thread
  - can continue while app is not active
  - Callbacks to forward results to service layer

- REST Methods
  - Generic service for REST communication (e.g. based on Google Volley)
  - implements GET/POST/... methods
  - prepares request entities
  - parses responses (e.g. JSON)
Example app: Twitter client
  - search and display tweets

```java
public class MainActivity extends Activity implements Receiver {
  private TwitterServiceHelper serviceHelper;
  public RESTReceiver mReceiver;
  ...
  public void onCreate(Bundle savedInstanceState) {
    ...
    serviceHelper = TwitterServiceHelper.getInstance(this);
    mReceiver = new RESTReceiver(new Handler());
    mReceiver.setReceiver(this);
}

  // invoke a call on ServiceHelper
  serviceHelper.twitterSearch(query, mReceiver);

  // get result via callback
  public void onReceiveResult(int resultCode, Bundle resultData) {
    ...
  }
```
- Singleton, simple asynchronous API at application level
- creates Intents and starts RESTService for each method
  - type of operation, URI, parameters, callback, request id
- implements Receiver -> callback passed to RESTService

```java
public class TwitterServiceHelper implements Receiver {
    ...
    public void twitterSearch(String query, ResultReceiver receiver) {
        Bundle extras = new Bundle();
        extras.putString("q", query);

        String reqId = String.valueOf(new Date().getTime()); // create id for request
        callbacks.put(reqId, receiver); // register request receiver in hashmap
        Intent intent = new Intent(context, RESTService.class);
        intent.putExtra("query", param);
        intent.putExtra(RESTService.RESULT_RECEIVER, receiver);
        intent.putExtra(RESTService.REQ_ID, reqId);
        context.startService(intent);
    }
```
REST Service

- extends IntentService -> REST call encoded into intent

```java
public class RESTService extends IntentService {
...
protected void onHandleIntent(Intent intent) {
    Uri action_uri = intent.getData(); // extract resource URI
    Bundle params = extras.getParcelable(PARAMS); // further parameters
    int verb = extras.getInt(HTTP_VERB, GET); // HTTP method
    // callback receiver
    ResultReceiver receiver = intent.getParcelableExtra(RESULT_RECEIVER);
    String requestId = extras.getString(REQ_ID); // request ID
    ...
    // invoke REST Method
    ...
    Bundle resultData = new Bundle();
    resultData.putString(REST_RESULT, EntityUtils.toString(responseEntity));
    resultData.putString(ACTION, action.toString());
    ...
    receiver.send(statusCode, resultData);
}
REST Method

- Prepares the HTTP URL & HTTP request body
- Executes the HTTP transaction
- Processes the HTTP response
- Based on java.net.URLConnection

```java
URL httpUrl = new URL(urlString);
HttpURLConnection httpConnection = (HttpURLConnection) httpUrl.openConnection();
httpConnection.setRequestMethod("GET");

if (httpConnection.getResponseCode() == HttpURLConnection.HTTP_OK) {
    BufferedReader in = new BufferedReader(new InputStreamReader(httpConnection.getInputStream()));
    String inputLine;
    StringBuffer response = new StringBuffer();

    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
    in.close();

    Bundle resultData = new Bundle();
    resultData.putString(REST_RESULT, response.toString());
```
REST - Implementation in Android
Example: Twitter Client

```java
twitterSearch(query, receiver)
```
Retrieving timeline entries

Problem with REST
- Heterogeneous client demand
- Over- and Under-Fetching
Demand-Driven Architecture

- First mentioned a QCon conference in 2015 by David Nolan and Kovas Goguta

- Three main principles:
  - Demand: Clients declare their data requirements (demand) by using a query language
  - Composition: demands for different features can be composed by using simple recursive data structures containing self-contained (sub-)demands
  - Demand is recursively interpreted by the service using a backend-agnostic interpretation layer
Server specifies available data with a schema (introspection)
- Client create individual schema-conform queries
- Server validates query against schema
- Resolver functions handle query fields to generate the result set, e.g. a resolver handles users and their attributes while another resolver handles activities
Example: GraphQL

- Specified by Facebook in 2012
- Since 2013 used by Facebook Apps on iOS and Android

```graphql
const User = { id: ID! // - required
  name: String
  timelines: [Timeline]
}

const Activity = { id: ID!
  name: String
  type: String
  timeline: Timeline
  media: [Media]
  participants: [User]
  activities: [Activity]

const Timeline = { id: ID!
  name: String
  owner: User
  participants: [User]
  activities: [Activity]

{user (id: 123) {
  name
  timeline (id: 1) {
    name
    activities {
      name
      type
    }
  }
}
```
Requests are specified by Client
- no static interface specified by server
- variable result set (e.g. subset of attributes and resources)

Composition of queries
- hierarchical queries
- cross-resource queries

Strong typing of schema and queries
- Server validates queries before processing

Storage-, Protocol- and Programming language agnostic

Stable query interface
- Extensible by schema
- No versioning required
• Distributing activities to multiple receivers
  • Push vs. pull

Server

User and Team data (workout, challenges, etc.)

Dr. Thomas Springer

Social Fitness App – Connectivity Challenge

Application Development - 5. Communication Mechanisms
- Basic Communication Scheme
- Asynchronous (without ACK)
  - no guarantee, that message has reached the Receiver
- Synchronous (with ACK)
  - Client and server have to be online at the same time
  - Sender is blocked for a short period of time only
Two roles:
- Supplier – creates messages
- Consumer – processes messages

- m:n Communication based on messages
- inherently asynchronous
- binding flexible based on subscriptions
- direct communication between supplier and consumer
**Loose coupling** - no direct message exchange between Supplier und Consumer

**Relation between suppliers and consumers created by:**
- Channel selection – get all messages from channel
  - subscribe(channelId)
- Subject/Topic – keyword used in subscription and messages
  - subscribe(temperature)
- Hierarchy – path in tree determines subtree, subscription for all nodes in subtree
  - subscribe(“de/sachsen/dresden/temperature”)
- Content-based
  - Subscribe(messageContains=“temperature AND Dresden”)

---

Dr. Thomas Springer

Application Development - 5. Communication Mechanisms
Layered Architecture

- notification – datum representing event
- overlay structure:
  - Event-based communication between Supplier and Consumer
  - Several underlying communication mechanisms

![Layered Architecture Diagram]

Dr. Thomas Springer  Application Development - 5. Communication Mechanisms  36
Mobility Issues

- **Physical mobility**
  - Suppliers/Consumers on mobile nodes
    - Location changes
    - Disconnections
  - transparent handover between brokers
    - adjustment of notification routing

- **Consumers**
  - Explicit resubscribe at new nodes
  - moveOut/moveIn operations for disconnect/reconnect
    - Not possible if client gets disconnected
    - Missing of notifications
    - Extended infrastructure support
      - heartbeats to detect disconnects
      - automatic reissuing of subscriptions after reconnect

- **Suppliers**
  - use advertisements
  - (advertise/unadvertise)
**Distributed Notification Routing**

- **Flooding**
  - Broker forward notifications to all neighbors
  - Only brokers with connected Consumers test filters

- **Filter-based Routing**
  - MessageBroker maintain routing tables with filter, destination pairs
  - Constantly updated based on subscribe/unsubscribe in system
  - Optimization by merging redundant/overlapping subscriptions
  - Trade-off: routing table size and effort for updates
Example: MQTT

- MQTT – Message Queue Telemetry Transport
- Topic-based
  - Each message has a topic, i.e. text descriptor
  - Hierarchically organized -> Subscriptions at different levels
Three Quality-of-Service Levels

- **At-most-once:**
  - The minimal level is zero and it guarantees a best effort delivery. A message won’t be acknowledged by the receiver or stored and redelivered by the sender. This is often called “fire and forget” and provides the same guarantee as the underlying TCP protocol.

- **At-least-once:**
  - When using QoS level 1, it is guaranteed that a message will be delivered at least once to the receiver. But the message can also be delivered more than once.

- **Exactly-once:**
  - The highest QoS is 2, it guarantees that each message is received only once by the counterpart. It is the safest and also the slowest quality of service level. The guarantee is provided by two flows there and back between sender and receiver.
Example: MQTT

- **Retained Messages:**
  - last message of a topic is stored by broker and automatically forwarded to new subscribers

- **Last Will and Testament (LWT):**
  - client can deposit MQTT message at broker
  - message is forwarded if broker detects unintended unavailability of client

- **Persistent Sessions:**
  - in case of frequent disconnections
  - Broker maintains persistent session storing session information and all messages for subscribers persistently
  - Reconnected clients do not have to subscribe repeatedly and get all missed messages forwarded
Example: MQTT

- Client-side library: Eclipse Paho
- MQTT Broker implementation: Mosquitto
- Android example client

MQTT Connect

```java
protected void onStart() {
    super.onStart();
    try {
        MemoryPersistence persistence = new MemoryPersistence();
        mqttClient = new MqttClient(
            "tcp://192.168.0.100:1883", "AndroidTest", persistence);

        mqttClient.connect();
        mqttClient.setCallback(this);
    } catch (MqttException e) { ... }
}
```
Example: MQTT

MQTT Callback

```java
public void connectionLost(Throwable cause) {
    Log.d("Main", "connection lost: " + cause);
}

public void messageArrived(String topic, MqttMessage message) throws Exception {
    Log.d("Main", "message received for topic: " + topic + ",
            message: " + message);
}

public void deliveryComplete(IMqttDeliveryToken token) {
    Log.d("Main", "deliveryComplete for token: " +
            token.getMessageId());
}
```
Example: MQTT

MQTT Subscription

```java
private boolean subscribe(String topic, int qos) {
    try {
        String topic = "topic/example";
        int qos = 1;
        mqttClient.subscribe(topic, qos);
    }
    catch (MqttException e) { ... }
}
```

MQTT Publication

```java
MqttMessage message = new MqttMessage(“Hello World”.getBytes());
mqttClient.publish("topic/example", message);
```
MQTT Disconnect

```java
protected void onStop() {
    super.onStop();

    if (mqttClient != null) {
        if (mqttClient.isConnected()) {
            try {
                mqttClient.disconnect();
            } catch (MqttException e) { ... }
        }
    }
}
```
Comparision of Interaction Schemes

Request/Response (RPC)

- close coupling (rebinding difficult)
- 1:1 communication
- inherently synchronous (problem of disconnections)
- familiar interaction scheme with call and result
- Client and Server are synchronized
- additional effort for reliability necessary (at-most-once-semantics)
- client/server-systems (data processing with results)

Publish/Subscribe (Message Queuing)

- loosely coupled (better support for rebinding)
- m:n communication
- inherent asynchronous (better handling of disconnections)
- simple interaction scheme
  - basis for more complex interactions
  - additional afford required to create complex interaction schemes
- explicit synchronization required
- no result for message
- reliable message exchange based on message queues
- load balancing, parallelization, batch processing, event distribution
- Bi-directional message exchange
  - E.g. chatting or exchange of timeline info.

![Cycling]

Communication Mechanisms

Server

User and Team data (workout, challenges, etc.)

27.3 km: 26:39
„Erste Etappe war eine einzige Quälerei, beim nächsten Mal geht’s bestimmt besser“
Summary

- **Request/Response Interaction**
  - Mobile RPC concept
  - RESTful services
  - Example: Google Volley

- **Demand-Driven Architectures**
  - Client-side queries
  - Example: GraphQL

- **Event-based Communication**
  - Publish/Subscribe
  - Event Channel
  - Example: MQTT

- **Bi-directional Communication**
  - E.g. Web sockets


Google I/O 2010 – Android REST client applications http://www.youtube.com/watch?v=xHXn3Kg2IQE


MQTT: http://mqtt.org