¡¡¡¡¡¡¡ HEAD

Chapter 1

Mobile client

In this chapter we talk about mobile computing.

1.1 Lab Description

The main objective of this assignment is to understand some of the some of the basic primitives and challenges of mobile computing.
You are required to expose Task managers Task entity as Google Endpoint (similar to the Todo item in the sample code) hosted on the Google App engine web application project, so that it can consumed by the mobile devises. Furthermore, also develop the functionality to send push notifications to the registered mobile devises whenever a new task is created on the app engine background. On the mobile application side, develop functionality for listing the tasks from the app engine background and also to display the push notification messages received from from the app engine backend whenever a new task is created on the app engine backend.

1.2 Solution

The solution implemented makes use of the Android Development Tool Bundle; a collection of programs and tools assisting creation of software run on the Android platform. Using the Eclipe-integrated software suite the majority of the code needed to complete the assignment can be auto generated. During the development the Google plugin for Eclipse is also heavily used.
Communication between the parts of the distributed solution is handled by Google endpoints. These are classes that exposes methods for receiving and modifying data objects. Each endpoint is expected to expose a single data entity, as defined by the javax.persistence namespace. The endpoint framework manages the communication between endpoints by converting method calls to REST requests and method return type to REST responses (see ref). Methods ar identified by use of signature attributes defined in the com.google.api.server.spi.config

namespace.
The solution consists of two eclipse projects: a backend and a client. The backend is deployed to the Google App Engine; a cloud service allowing for web services communicating via Google endpoints to be deployed and run on Googles dedicated servers. The client is an Android application communicating with the backend by use of Google endpoints.

1.3 Example Run

1.4 Theory

Mobile computing (the connectedness of devices moving about in physical space) and ubiquitous computing (the integration of computanional devices in physical space in everyday life) are, in this context, essentially the same in that they are both volatile i.e., change is the rule rather than the exception. As they move about physical space these computanional devices connect and disconnect to each other and the surrounding computational infrastructure. Mobile computing takes place *between* physical spaces where ubiquitous computing is embedded into the physical space.

Because of this, mobile computing is considered 'volatile'.

Volatility

Volatility includes: failures of devices and communication links. Changes in bandwith (communication charateristics). Frequent creation and destruction of associations (between components).

Mobile and ubiquitous computing exhibits all of the above forms of volatility due to their integration into(and thus dynamic relation to) the physical world around them.

Smart Spaces

A smart space is any physical place with embedded devices in them. Mobile devices can enter and leave smart spaces in 4 ways.

- Physical movement: a *device* may enter or leave a space.
- Logical mobility: a *process* may enter or leave a space.
- Static devices may enter or leave the space e.g. a user may add or remove-processes to her smart phone.

- Devices may fail i.e., dissapear from the space.

However, the only matter of concern is whether a device/ process changes it's association with other processes i.e. a device either appear or dissappear from a smart space. Thus, it is irrelevant if a device /process is recident or visiting. Note some distinction is made as to the characteristics of connectivity in smart spaces e.g. the rate of association-change, or security issues (it matter which devices move in or out of a secure space).

1.4.1 Device Model

Ubiquitous and mobile computing leads to smaller and smaller devices which in turn puts restraints on energy supply and computing resources. This, taken into account in the following model.

Energy: The smaller the device (assuming only battery powered devices) the lower the battery capacity. This increases the propability of device failure and it poses a challenge to the frugality of algorithms applied.

Resource constraints: To make mobile and ubiquitous device physically small enough to embed in the physical space (e.g. to carry around), they naturally have less resources i.e processor speed, disc space and network bandwidth. How do we design algorithms that execute in a reasonable time and utilize resources in the surrounding space?

Sensors and actuators: To make devices context-aware (integrated in, and interaracting with the physical space) they are fitted with sensors and actuators e.g. a mobile phone may have a GPS fitted. In order to become ubiquitous these sensors are mass-produced, cheaply. Can we then trust them to function correctly/precisely ? see P.839.

Connectivity: Devices have one or several wireless connectivities e.g. bluetooth, WiFi etc. The volatility of the connectivity has an impact on system properties e.g. Wireless disconnections are more likely than wired disconnections. A failure to process a request may be due to missing components somewhere else in the interconnected system. P. 841

Spontaneous interaction: Components routinely connects and disconnects amongst themselves. we say that *Associated* devices *interact* i.e., devices can be associated without actually interacting. This poses a problem with security. What trust(privacy) can there be when devices spontaneously interacts and are spontaneously connected? E.g. if a sensor automatically registers users

within its space and the users' other devices spontaneously interacts with other devices in this space this provides the opposite of security and privacy. The correlation between transactions on a creditcard and the movements of a user could reveal information. this is not a secure situation.

None the less, devices need to interoperate without the users' knowledge . This happens in two ways:

- Network bootstrapping: A server within a smart space provides an IP address and DNS parameter which devices queries. The server broadcast or multicast these informations.
- Association: When connected to a space, how do the device connect to the right process among possibly many ? and how do we constrain the communication to only devices within that particular space?

  The boundary principle states that a space should define its boundaries as systems boundaries, not association boundaries (components from outside the space may connect to components within but the space may provide only the services belonging to it.)

On way to solve the association problem is to use discovery services.

1.4.2 Discovery Services

A discovery service is a directory service that takes into account volatile system properties [see section **??**volatility]. Device discovery and service discovery services exists. In device discovery the user selects among the available devices and queries that device for services offered. Service discovery happens automatically in situations where the user is concerned with the properties of the service, only.

Discovery services has interfaces for *registrering* (with a given address and attributes) and *deregistering* services (note devices may dissappear without deregistering as per the properties of volatile systems!) and for *lookup* of services. Each service matching a clients requests is returned with its address and attributes. Chosing a service is done by a secondary call (by the user, or automagically).

Serverless Discovery

Bootstrapping access to a local discovery service usually involves broadcasting(or multicasting) queries to a known IP address. Note this address must be know a priori by all devices. An alternative is to use *serverless discovery* where devices collaborate to provide the directory service. There are two models. *Push* and *Pull*. In the push model services regularly multicast their descriptions. This

comes at a price on bandwidth and energy use as devices continually push and listens for pushs. In the pull model clients multicast their queries and services respond (if their descriptions match). In this model there's no wasted bandwidth but the client may receive several responses to a single request where one would do.

A device may leave without deregistering with a discovery service but the service needs to know as soon as possible, to maintain a fresh state. This can be achieved with *leases*. A lease is an allocation of a resource with a build-in deadline., The client can renew the lease by repeating its request before the deadline (E.g. by calling a *refresh()* method). Otherwise the service may reallocate the resource. Leases apply to services as well. Note there's a tradeoff between lease period and bandwitdh consumption.

Remains; the issue of scope. How do we define the boundaries of a smart space? ways include having the user identifying himself to the service physically or by means of glyphs on a device. Other ways inlude physically constrained devices (devices whose reach are limited in space.)

Real World Example:

Android's 'discoveryListener' (a callback handler for registering services) involves handling a 'onServiceFound()', 'onServiceLost' and 'onServiceFailed()' methods.

The andriod NSD API then handles when services are found and lost etc.

Likewise the android system uses a ResolveListener callback to handle callbacks from the resolveService() method. The callback includes service information including an IP address and port number.

To preserve resources applications pause their discovery services when they become inactive and start them back up when they become active again. This is done with the onPause() and onResume() methods. Applications deregister with the DNS on exit by calling onDestroy() and tearDown() methods.

1.4.3 Interoperation

Enough about connecting devices. Now we look at how devices interoperate.

The problem is heterogeneity. In a volatile environment the difference in component interfaces makes for a challenge. We seek to give devices a reasonable chance of interoperating with each other in, and between, spaces (the benefits of mobility can't be fully utilized if each space has its own programing

language). One approach is to allow heterogeneity, and to compensate for differences in interfaces by using adapters. This could potentially yield more than a lot of adapters. Another way is to try to constrain the syntax of the interfaces. Off course, using a single interface is much easier (and resource-friendly) than having to adapt to an ever increasing and varying number of different interfaces. This use of a simple unvarying interface is called a *data-oriented* system. Data-oriented systems trade agreements on a set of functions for agreement on the types of data it accepts (the internet as a paramount example of a data-oriented system i.e., the HTTP protocol has limited interface...).

Note a system with a fixed interface still has to check compatibility between to components by means of either meta-data or by verification of data sent to it.

Interoperations Models

Two ways of interoperation in volatile environments are events and tuple spaces. The subscription style communication in event based systems is clearly suitable for volatile systems but note that even here components can only operate correctly if they agree on the service and on the attributes and types of events i.e., they need agreements on the syntax and semantics used.

Aside, and to much amusement to the authors, XML does not provide a solution to the problem of interoperation in volatile systems, the problem of syntax and semantics. It merely provides a self-describing data structure. HaHaHihi giggle giggle, hrmuphs (hate XML).

Sensing And Context-wareness

Apart from the volatility of mobile and ubiquitous computing, there is the matter of integration into the physical world, and context-aware systems. i.e., software fitted to sensors and how they respond to the sensed data. E.g. a cars brakes reacts differently when it is cold outside).

There are four challenges to context-aware systems.

- Physical constraints (how to fit the sensor in the environment?).
- Abstraction from data (the same type of sensor may produce different data depending on the use.) The application need som agreement from the sensors of the meaning of their data output.
- An application might need to combine sensor data to produce a reliable output (e.g. combine a camera for face recognision with a microphone for voice detection.)
- Context changes (e.g. the device changes location spontaneously)

1.5 Conclusion