

Programming Ubiquitous Things
2024/2025 - Spring 2025 - UiO

Adaptability

Prof. Paulo Ferreira
paulofe@ifi.uio.no
UiO/IFI/PT

1



Contents

- Definition of adaptability
- Examples of scenarios
- Variable context/resources
- Trading-off adaptability and invisibility
- Measuring/Evaluating the effectiveness of adaptability mechanisms
- Adaption layer (operating system, middleware, application)
- Architecture for adaptability
- Adaptability policies
- Access to objects (replication, consistency, transactions/sessions,failure)

2

2

Definition of Adaptability

- **Adaptability** (Latin: *adaptō* "fit to, adjust")
  - is a feature of a system or of a process
  - quality of being adaptable; a quality that renders adaptable
  - variability in respect to, or under the influence of, external conditions
- A system is said to have the capability of being adaptable:
  - if it **adapts automatically** to variable working conditions,
  - while offering the **best quality** of service possible,
  - that is **acceptable by the user**

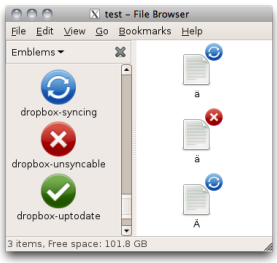
3

3

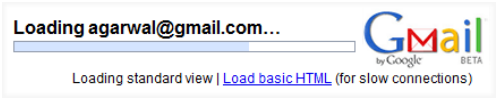
Examples of Scenarios

- In the train watching a video
- Accessing files from a laptop with GSM/3G/4G availability
- Reading email on a slow network connection









4

4

Variable Context/Resources (1/2)



- Due to their intrinsic nature, execution environments in mobile/ubiquitous computing suffer from **great and diverse variations** during applications execution
- These variations can be:
  - **qualitative** (e.g., network connection or disconnection, specific devices such as printers in the device neighborhood, consistency and security constrains), or
  - **quantitative** (e.g., amount of usable bandwidth, memory available)
- Applications must be able to deal with this variability
- Applications programmers should not be forced to **account for every possible scenario** in their coding as this would be:
  - inefficient, error-prone, and limited to situations accounted for *a priori*

5

5

Variable Context/Resources (2/2)



- Which **resources should the system adapt to** ?
  - system: network, energy, CPU, memory, etc.
  - environment: luminosity, temperature, etc.
- An adaptable system must consider:
  - **all resources** which quality/availability **may change** (with time/location/etc.) while the **application is running**

6

6

Trading-off Adaptability and Invisibility

- **Correct adaptability is key for invisibility**

- If a systems does not adapt at all:
  - it stops working
- If a system adapts wrongly:
  - it malfunctions

- **Both cases make the system visible !**

- Being adaptive implies:
  - **detect** changes and **act** accordingly
  - **predict** changes to allow timely adaptation

How can we measure/evaluate the effectiveness of adaptability mechanisms ?

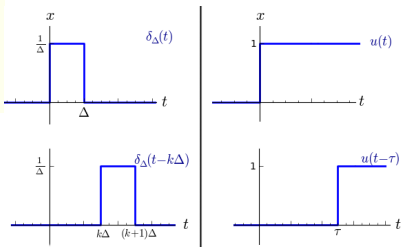
7

7

Measuring the Effectiveness of Adaptability Mechanisms

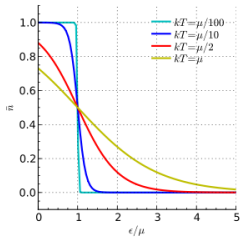


- User satisfaction



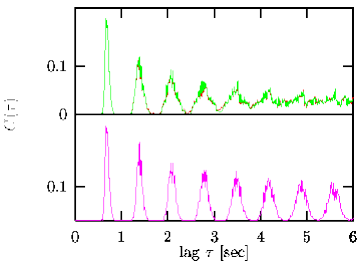
- “Dirac/Spike” changes on resources availability

- Speed to react



- System stability

- Portability of the solutions



- Resource usage of the adaptability solutions

8

8

Adaptation Layer (operating system, middleware, application)

- At what **level** should the adaptability solution be implemented ?
- **Operating system**:
  - oblivious to application semantics/no application awareness
  - global/equal solutions to all
  - implications with OS portability
- **Middleware**:
  - common mechanims can be reused
  - interaction with applications to be semantic aware
  - portable
- **Application**:
  - semantic aware
  - specific for each application
  - not portable for different applications

Is there a best solution?

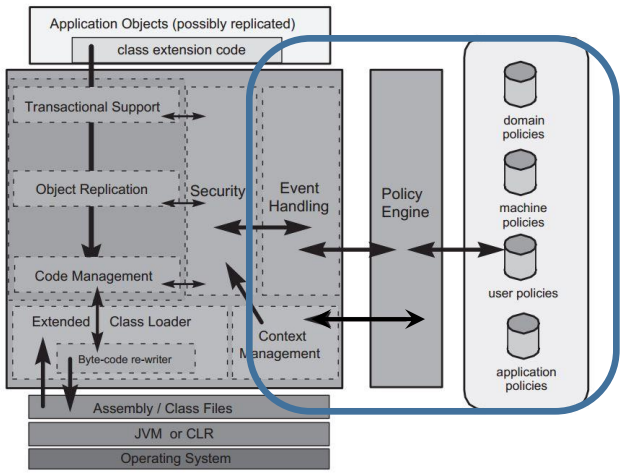
Is there a one solution fits all?

9

9

Architecture for Adaptability (1/3)

- Dealing with such **variability** can be achieved through **automatic adaptation of applications**
- Thus, the middleware must provide:
  - **flexibility for application development**, and
  - **runtime adaptability**
- Then, applications can cope with the multiple requirements and usage diversity found in mobile settings

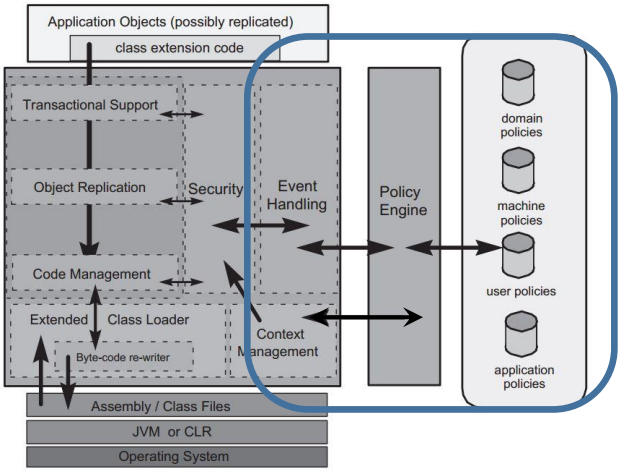


10

10

Architecture for Adaptability (2/3)

- Policy engine:
  - it is the inference component that **triggers or mediates** responses to events occurred in the system
  - it holds a **variable set of policies** to be enforced in the system
  - it receives events generated by middleware modules and applications, evaluates **policy rules** and triggers events, handled by actions based on evaluation results
  - e.g, object replication is performed according to specified **policies**
- Context management:
  - performs **resource abstraction** and manages properties whose values vary during execution

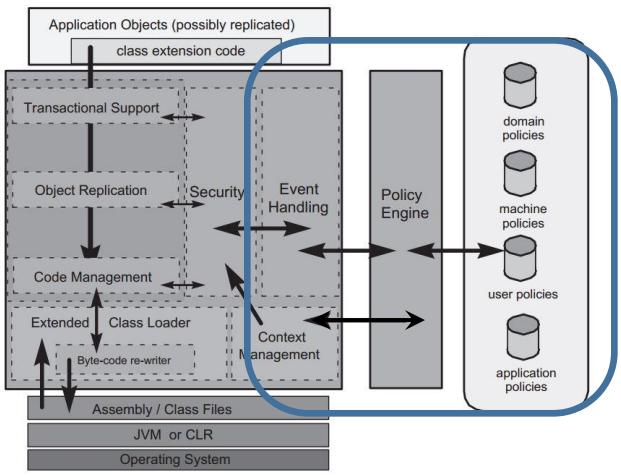


11

11

Architecture for Adaptability (3/3)

- Situations handled by the context management:
  - **appearing** devices, **discovering** remote resources or application counterparts
- Detecting these situations allows the middleware and the applications to **decide**:
  - e.g., whether to **replicate** data from different sources, **swap** some data out, etc.
- In general terms:
  - any **change** to the properties (resources, middleware state or user-defined properties) managed by the context manager
  - can potentially **trigger** associated events defined by the **policies loaded**



12

12

Adaptability Policies (1/2)



- Adaptability can be provided through the enforcement of **declaratively-defined policies** supported by the middleware:
  - **policies are not hard-coded** in applications, and
  - can be **deployed, enforced, and updated at any time**
- Such middleware relies strongly on the following features:
  - the extensible capability to support the **specification and enforcement** of runtime management **policies**,
  - a **set of pre-defined policies** to control the mechanisms previously mentioned
  - e.g., a pluggable set of basic mechanisms supporting object replication

13

13

Adaptability Policies (2/2)



- We address the issue of **adaptability** specific for **replication** and **transactional/session** access to data:
  - these are two fundamental aspects regarding how the middleware adapts to variability in mobile settings
- This results in **policies** handling:
  - **consistency, fetching, delegation, atomicity, caching, failure**
- For this purpose:
  - there must be a **set of pre-defined policies** and **policy-driven modules**,
  - provided to applications programmers,
  - to manage specific execution mechanisms (replication and transactions/sessions, in particular)
- Such **policies** control **which objects are accessed** by applications under certain circumstances:
  - e.g., **which replica(s)**, **from which node(s), within (a) transaction(s)/session(s)**
  - the set of objects that are accessed within **transactions/sessions** has obvious impacts on the **consistency** of the data as well as on its **atomicity**
  - the **correctness** of the data accessed depends on the applications semantics which is well-known by the programmer

14

14



Examples of Policies

replication and transactions

15



Replication

- Object replication is **incremental** and **adaptive**
- Unless otherwise specified, it is performed transparently to applications but can also be **flexibly configured**
- In particular, the mobile middleware must allow the specification of the following:
  - the **best moment to create a replica**,
  - **when/how to merge** two or more replicas of the same object,
  - **how many objects to replicate** at a given time (a cluster of objects),
  - which **branch of a graph should be further replicated**,
  - which objects should be **swapped-out** (i.e., dynamically replaced by a proxy-out and transfer the remaining objects to a neighboring device)

16

Transactions/Sessions - Access to Objects (1/2)



- Given that mobile environments are highly dynamic:
  - nodes **enter** and **leave** networks,
  - communication **links are more prone to failures**, and
  - **disconnections** are frequent.

No longer typical transactions. We call it sessions.

- This leads to **distributed transactions** running on these environments to **abort unnecessarily:**
  - due to connectivity problems **even though conflicts do not exist** between transactions
- Strict transactional solutions have some important drawbacks leading either to **unnecessary aborts** or to **unnecessary reduced data availability**
- Typically, if a transaction is not able to **commit**, it is forced to **abort**:
  - in most transational proposals, **alternatives** instead of aborting the transaction are not provided, or
  - the existing alternatives are neither **flexible** nor **simple** enough to program by **application developers**

17

17

Transactions/Sessions - Access to Objects (2/2)



- The requirements of applications regarding transaction properties can be very different:
  - require **strict atomicity**,
  - **discard some changes** and still **commit** successfully
  - **abort** the transaction immediately **if some node is not reachable**
  - **postpone** the commitment of the transaction

No longer typical transactions. We call it sessions.

- Different degrees of control (e.g., consistency) may be required by developers/application:
  - some developers may **not desire to configure every detail** of the transaction behavior,
  - while others **may even want to react to the changes of the environment in real-time**
- Finally:
  - it is **complex to specify the transactions/sessions behavior** because it is necessary to think about all the situations and problems that can occur
  - in addition, if we need to adjust the transaction/session properties, it is necessary to **change the code of the transaction/session**
- **Transactions/session behavior** can be **adapted in run-time** to deal with the **specific scenarios** and **applications needs**:
  - replication and transaction **policies focus on dealing with disconnection**, and
  - on specifying the **minimum consistency and atomicity properties** of a transaction

18

18

Policies (1/2)



consistency, fetching, delegation, atomicity, caching, failure

- Consistency:
  - allow the usage of **outdated versions** of objects if the corresponding **remote node is not available**
  - this can be done **per object or per set of objects**
- Fetching:
  - describes if (and how many) objects must be **pre-fetched before** executing the transaction or if objects should be **fetched on demand** while the transaction executes
- Delegation:
  - when a transaction/session is about to be committed, it is possible to delegate that functionality, i.e., to **transfer the commit responsibility to some other node**

19

19

Policies (2/2)



consistency, fetching, delegation, atomicity, caching, failure

- Atomicity:
  - specify if the transaction/session can **commit even if not all nodes involved in the transaction are reachable**, i.e., if some changes can be dropped
- Caching:
  - it is possible to store both the fetched objects and the committed objects by local transactions/sessions, in a node's **cache**
  - this feature is essential for providing **access to data during disconnection**
- Failure Handling:
  - how to react when the specified **conditions** of consistency, fetching and atomicity **do not hold**

20

20

Conclusions



- **Adaptability** is a key issue to support **invisible computing**
- The **level at which support of adaptability is provided** is a key aspect (OS, middleware, application)
- **Policies, not hard-coded**, are fundamental:
  - consistency → - it is impossible to access the most up-to-date replica of an object, but **it is possible to obtain a replica slightly out-of-date** that is still enough for the application to proceed
  - fetching → - **before the transaction/session starts executing, or on-demand**, in which case, objects (sub)graphs are only fetched upon the first access, during the transaction/session
  - delegation → - delegate the **responsibility of commitment** to another node, i.e. the role of coordinator will be performed by some other node
  - atomicity → - specify **which objects must be updated during commit**, and **which objects do not cause the transaction/session to abort** in case it is not possible to commit them successfully (e.g., the home node is not online)
  - caching → - store copies of objects that belong to the transaction/session dataset in the **local cache**
  - failure → - how such **failures should be handled** (consistency, fetching, etc.)

21

21