DMPP: main concepts

Bruno Rocha Pereira

June 15, 2016

Chapter 1

Introduction

- **Deutsch fallacies (you don't need to know the list by heart but to understand what each fallacy implies)**.

  In a perfect world:

  1. The network is reliable!
  2. The network is secure!
  3. The network is homogeneous!
     → All devices have the same configuration, ports and OS
  4. The topology does not change!
  5. Latency is zero!
  6. Bandwidth is infinite!
  7. Transport cost is zero!
  8. There is one administrator

- **Distribution vs concurrency, distribution vs. mobility**.

  Distributed systems are concurrent programs with no resource shared. Parallel programs can be used with only one core while still having concurrency problem.

  Mobile as in mobile phone. Can join and leave the network at any time. Distributed as in using a system with devices on the same network sending each other messages.

- **Distribution transparency and its fundamental issues (again, you don't need to know the list of problems by heart but to understand what each issue implies and why distribution transparency is said to be a myth).**

  - Latency: There can be a delay between sending a message and the moment when the other device gets it.
  - Difference in Memory Access : Every method invocation implies sending information over the networ to retrieve the reference.
  - Partial Failure: An element of the sytem can fail

  - Concurrency: automatic parallelisation cannot be made automatic because of partial failure and latency

- **Language vs. middleware vs. reflective approach.**
  - Language : concepts create a language, hiding the details
  - Middleware : library, makes use of another language's functionalities.
  - Reflective Approach : use of Aspects or reflection to combine both

- **Essential complexity vs. accidental complexity.**
  - Essential complexity: complexity bound to the problem. Cannot be avoided
  - Accidental complexity: bound to the development of the solution. Can be avoided by the dev.

Chapter 2

Prototype-based Programming

- **Why are prototypes relevant for distributed programming?**

  Classes weren't really suitable because they need to be sent everytime over the network (which can fail). Prototypes (with objects and no classes) make sharing explicit.

- **Symbiosis: understand the concept, how symbiosis works between AmbientTalk and Java.**

  All java code can be accessed from AT but layers are different (Java is class-based while AT is prototype-based) so a wrapping with an interface is necessary. Although AT doesn't have overloading, Java has it and it can be used. Casting must be used to avoid ambiguity.

- **Object creation exnihilo vs. cloning, cloning and instantiation, delegation and cloning.**

  - Ex nihilo : done with `object:`
  - Cloning: makes a shallow copy with `clone:` or by creating a constructor method with `init` and use `Object.new()`.

- **Scoping: lexical vs object scope,methods vs. closures.**

  - Lexical scope : scope of the code
  - Object code : scope of the object

  Closures can be seen as lambdas and are located inside methods

- **Forwarding vs. delegation.**

  - Delegation: Used with the keyword `extend:`. Can be explicit with `super^` or implicitely thanks to the method lookup. The operator m̂eans a delegation. Allows a late binding to `self`
  - Forwarding

- **The uniform access principle (UAP), UAP and closures, first-class methods.**
  - Uniform access principle: Getters and Setters are implicit. Reading a variable invokes the method with the same name.
  - UAP and closures
  - First-class methods: Methods are first-class citizen, which means they can be given as parameters as other methods. When selecting a field from an object, the resulting closure is an accessor for the field, i.e. a nullary (no argument) closure that upon application returns the field's value.

- **Classifying objects with type tags.**

  Objects can be tagged with a type. First the type needs to be define ith `deftype`, then a object can be tagged with `taggedAs:`.

- **The concept of traits, objects as traits.**

  In objects, we can `import:` another object that is considered as a trait. A trait is a set of methods and attributes that will be inserted in the class that imports it. All the methods of the trait can be then used thanks to AT delegation support.

Chapter 3

Event-loop Programming

- **Threads: understand the model, how to achieve inter and intraobject synchronization, understand why and how deadlocks happen.**

  Trivial

- **Actors: the basic principles of the model, and difference with the thread model.**

  Actors are concurent entities that can send and receive messages. Those messages are synchronous and are stored in mailbox

- **Active Objects: understand the model, and its limitations for distribution.**

  Has a passive object that is contained in active objects. The problem is that the passive object is "private" and can't be accessed from another active object. Other actors have no idea about the internal state of an actor.

- **Event Loops: understand the model, the concept of far references, and the three concurrency control properties that the model can enforce.**

  As opposed to Active objects, passive objects can refer to each other via near reference (from the same object) and far reference (from a different one).

  - Serial Execution Property: No race condition, events are processed one by one
  - Non-blocking Communication Property: No deadlocks, an event-loop never suspends
  - Exclusive State Access Property: No race conditions, you will never have 2 threads trying to access the same resources

- **Actors in AmbientTalk, asynchronous message sending and parameter passing rules.**

  Sending asynchronous messages can be done to either far or near references using the $\leftarrow$. They are sent to the mailbox of the concerned actor.

- **Asynchronous message sending and return values:**

  Async messages have no return value.

- **The concept of customer objects, and its limitations.**

  Customers objects allow to receive an answer from sending a message to an actor. The problem is that it needs to be given as a parameter to the method.

- **Futures: the concept, blocking vs. nonblocking futures, future pipelining, how to achieve conditional synchronization with futures.**

  Futures are computation that take time to be computed on a different thread. They can be waited for (Blocking) or not (Non-blocking). Futures in AT can be `unresolved, resolved` or `ruined`. Listeners `when:<future> becomes:` can be applied around it. The closure with a parameter following it is computed when the future gets resolved to a value.

  Futures can be pipelined when a computation needs to be done on the result of another future. Instead we can use `when: actor<-m1()<-m2() becomes:`

Chapter 4

Event-based Distributed Programming

- **Understand the different design issues to provide a distributed object model, and its relation to the different types of networks.**
  - Discovery: How can objects get to know each other on a network?: You don't know by default who is on the network, they have to discover each other first.
  - Communication: What is the communication model between two distributed objects?: They need to agree with each other on how to communicate
  - Synchronization: How do distributed objects coordinate their actions?: synch/asynch?
  - Failure Handling: How do distributed objects deal with (presumably)failed peers?: network can be unreliable
- **Know the discriminating properties of mobile networks, and understand the requirements for a distributed object model for MANETs.**

  Mobile Ad-hoc networks have no infrastructure like servers, have volatile connection (devices can join and leave at any time). Failures can happen at any time.

  MANETs need:
  - Decentralised Service Discovery Protocol: There is no infrastruture, they need to discover each other by themselves. Actor model needs however to know the methods of a named actor.
  - Decoupling in Time: They don't need to be connected at the same time since they can join and leave at any time.
  - Decoupling in Space: Devices do not need to know each other beforehand, they can start communicating even though
  - Connection-independent Failure Model:

- **Far references and partial failures: how far references deal with intermittent failures.**

  When a failure occurs, a far reference stores all the messages sent. This buffer is emptied when the object reconnects. This allows the Time Decoupling.

- **Ways of obtaining a first far reference, service discovery vs. service lookup.**

  A far reference can be otained by receiving it from another object or by discovering the object.

  - Service discovery: Devices discover each other in a spontaneous way.
  - Service lookup: Can be seen as a DNS, an object is dedicated to storing all the far references -¿ not adapted to MANETS since they require a kind of central server.

- **Distributed object scoping and isolates.**

  Object are always passed by far reference. Isolates however are objects with no lexical scopes. They are passed by copy.

- **Far references and permanent failures, leased object references.**

  When we want to delete far references in the case of a permanent failure, we can lease them. That means associating a timer to the far reference. When the time is elapsed, the far reference can be garbage collected.

- **Understand the difference between pessimistic and optimistic distributed object model.**

  - Pessimistic: We're aware that the object can disappear
  - Optimistic: We consider that the object can last forever.

Chapter 5

Meta-level Engineering

- Metaprogramming: firstclass messages, quasiquoting and splicing, first-class abstract grammar
- Reflective programming: base vs meta level, meta vs. reflective program.
- Terminology on reflection (reification, introspection, intercession), structural vs. behavioural reflection.
- The concept of meta objects, and understand the problems with popular meta-level architectures
- Mirror-based reflection: understand the model and the three properties that the model can enforce.
- Mirrors in AmbientTalk: explicit vs. implicit mirrors on objects.
- Mirages: the concept.
- Mirrors on Actors: understand which operations actor mirrors reify in comparison to object mirrors.
- MOP: You are not expected to know by hearth the message invocation protocol but the overall idea and understand the relevant parts that need to be altered in order to implement language constructs like futures and leased references.

Chapter 6

Coordination using Tuple Spaces

- Terminology: coordination, datadriven vs. control driven coordination.
- Tuple Spaces: the basic interaction mechanism, understand the Linda model and how tuple matching works.
- Know what decoupling in time and space, and synchronization decoupling is, and being able to reason about the forms of decoupling that tuple spaces and other communication paradigms seen at the course exhibit.
- Federation of mobile tuple spaces: understand the basic interaction model of tuple spaces in a mobile setting, and being able to explain the two variations (LIME & TOTA), know what are the differences and similarities of the two variations.
- TOTAM: understand the goal for this hybrid approach, and key features, and how the model supports memory management in face of failures.

6.1 Decoupling

6.1.1 Space decoupling

The interacting parties do not need to know each other. The publishers publish events through an event service, and the subscribers get these events indirectly through the event service. Publishers and suscribers do not hold references to each other neither do they know how many of them are participating

6.1.2 Time decoupling

The interacting parties do not need to be actively participating in the same interaction at the same time. Events occuring when a subscriber is offline are delivered when he gets back online.

6.2 Tuple space

6.2.1 Primitives

out an actor outputs a tuple into the tuplespace

peek an actor reads matching tuples

in an actor grabs a tuple into itself

6.2.2 LIME (Linda In a Mobile Environment)

*Put more intelligence in the tuplespace **operations** (in, out, peek, ...)*

This is a federated tuple space model (make everyone agree). Hard to garantee consistency: atomicity only for `in` operations. Implies tradeoffs for decoupling in space and time.

6.2.3 TOTA (Tuples Over The Air)

*Put more intellignes in the **tuples** themselves*

A replication-based tuple space model. Fully decoupled in space and time, but atomicity cannot be garanteed.

- Tuples are injected into the network, and hop between spaces.
- Each tuple carries a propagation and a maintenance rule; ie `Tuple = (Content, propRule, maintRule)`
- `in/peek` only affects the local tuplespace and direct neighbors

Supporting Tuples and safe state

`T1` in a space `S1` is a **supporting tuple** of `T2` in a space `S2` if:

- They share the same key
- There is 1 hop between `S1` and `S2`
- `T1.hop = n` and `T2.hop = n+1`

A tuple is in **safe state** if it has at least one supporting tuple, or it is a source tuple (hop = 0) A tuple **deletes itself** if it is no longer in a safe state

6.2.4 TOTAm (Tuples Over The Ambient)

Mixed approach of LIME and TOTA. Atomicity for `in` operations. `read` decoupled in time. Scoped propagation protocol. Reacts on incoming tuples.

Private and public tuples

The classical out operation acts on the local tuplespace. `inject` into the ambient. Each tuple carries a propagation protocol and a sign. To remove tuples, inject an **antituple**: a tuple with the same content and propagation rules, but opposite sign.

Chapter 7

Peer-to-peer systems

- Typical characteristics from P2P systems.
- The concept of an overlay network.
- First generation of P2P systems: motivation and limitations.
- Second generation of P2P systems: understand the basics, flooding, TTL propagation
- Understand the motivation for P2P third generation and which guarantees they provide w.r.t. previous generations, distributed hash tables
- Chord: understand how keys are distributed, the lookup algorithm and the different cost models of lookup, how join and leave of nodes affect the network, and the role of periodic stabilization
- CAP theorem: understand consistency, availability, partition tolerance means and their tradeoff. Be able to reason which guarantee Chord and other distributed algorithms or applications cannot provide.
- Distributed storage: know why P2P systems scale well specially for immutable objects, being able to reason about different replication strategies.
- Beernet: the goal of the approach and basic architecture. Understand the differences with Chord with respect to management of peer failures, and why it is relevant in a mobile setting.

Fundamentals

- Understand why external synchronization with physical clocks is not employed for coordinating distributed processes.
- Lamport's clocks: the algorithm, know which properties Lamport's clocks exhibit, understand what it means that Lamport clocks are not strongly consistent.
- Partial vs. total ordering of events.
- Vector clocks: Understand the difference with Lamport's clocks, which properties they exhibit, and how you can compare vector clocks. Understand the limitations and assumptions of vector clocks.
- Be able to reason about application domains where lamport and vector clocks can be applied.
- Consensus: two Generals' problem, know what the consensus problem is.
- Know what it means the safety and liveness properties, and being able to reason about those properties for distributed algorithms.
- Know the FLP Theorem and understand its implications.
- Paxos:
  - understand the goal, the roles of nodes and the purpose of each phase.
  - Be able to explain why the algorithm cannot reach agreement sometimes, and how the safety properties are guaranteed.
  - Be able to reason about application domains where paxos is useful.

Chapter 8

Distributed Programming on the Web

- Know the basic interaction model with a client-server architecture communicating by means of HTTP request.
- Understand why programming on the web 1.0 is said to be programming with strings.
- Know what a RIA is, and the role of JavaScript in the development of RIAs.
- AJAX: understand the idea that scripts can asynchronously communicate with the server, and why the interactions run asynchronously.
- Thread-based vs. event-based servers.
- Know the two ways how to achieve communication amongst clients (Publish/ Subscribe on top of HTTP and websockets) and reason about their advantages and disadvantages.
- Mobile cross-platform solutions:
  - The idea of employing web-based technologies (i.e. JavaScript) for mobile computing, and which advantages brings to mobile software development.
  - Understand the basics of two big family of cross-platform solutions (hybrid vs. interpreted) and be able to reason about their advantages and disadvantages.

Chapter 9

Reactive Programming

- Understand the differences between classic sequential and event-driven software.
- Know the advantages of event-driven client and servers for RIAs.
- Know what the issue of "Inversion of Control" (a.ka. the callback hell effect) is, and its implications (w.r.t shared state)
- Reactive programming: the basic model of evaluation, and key abstractions (event sources vs. behaviours), lifting, glitches and the most common dataflow evaluation strategy to reevaluate code without causing glitches.
- Flapjax:
  - understand how pages can be made reactive
  - you do not need to know all the operations on event stream combinators, but you should be able to reason about a piece of code with respect to which parts are reactive, the dependency graph, and how reactive code communicates with the DOM and server).
  - Understand what it means that there are no callbacks in Flapjax in contrast to JavaScript.