

# A Note on Distributed Computing

- The Waldo et al paper is a high-level manifesto for a distributed computing programming methodology

# Overview

- Main thesis of the paper: *distributed* computing is very different from *local* computing
- We shouldn't be trying to make one resemble the other
- We cannot hide the specifics of whether an object is distributed or local (“paper over” the network)
- Distributing objects *cannot* be an afterthought
  - there are often dependencies in an object's interface that determine whether it can be remote or not
- The “vision of unified objects” contains fallacies

# Vision of Unified Objects

- What is it?
  - Design and implement your application, without consideration of whether objects are local or remote
  - Then, choose object locations and interfaces for performance
  - Finally, expand objects to deal with partial failures (e.g., network outages) by adding replication, transactions, etc.
- This paper argues that the premise is wrong:
  - the design of an application is dependent on whether it is local or remote
  - the implementation is dependent on whether it is local or remote
  - the interfaces to objects are dependent on whether objects are local or remote

# Differences between local and distributed computing

Latency, memory access, partial failure, and concurrency

- Latency: remote operations take much longer to complete than local ones
- Memory access: cannot access remote memory directly (e.g., with pointers)
- Partial failure and concurrency: remote operations may fail, or parts of them may fail. Also, distributed objects can be accessed concurrently and need to synchronize

# How do differences affect programming?

- Latency:
  - if ignored leads to performance problems
  - important, but critical?
  - can be alleviated with judicious object placement
- Memory access:
  - Waldo notes that it would be too restrictive to prevent programmers from manipulating memory through pointers
  - things have changed a lot. Java papers over memory *and* makes everything be an object. Hence, it's all a matter of defining the right abstractions
- Partial failure and concurrency:
  - more serious problems, as operations fail often, and sometimes parts of them succeed and cause later trouble
  - this is an important factor!

# Dealing with partial failure

We can either

- treat all objects as local objects

or

- treat all objects as distributed objects

Problems:

- The former cannot handle failure well
- The latter is a non-solution: instead of making distributed computing as simple as local, we make local computing as hard as distributed
- The same holds for concurrency!

# Some examples

- Imagine a “queue” data structure object
  - interface:
  - enqueue(object), dequeue(object), etc.
  - the queue is held remotely
- Problems:
  - on timeout, should I re-insert?
  - what if insertion fails completely?
  - what if insertion succeeded but confirmation was not received?
  - how do I avoid duplication?
  - need request identifiers, but the queue interface does not support them!
- In short, recovery from partial failure cannot be an afterthought. Implementation choices are apparent in the client interface. No “ideal” interface is suitable for all implementations.
- Same for performance (example of set and testing object equality)