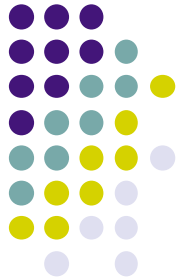
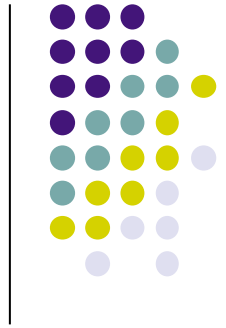
Deterministic dataflow summary



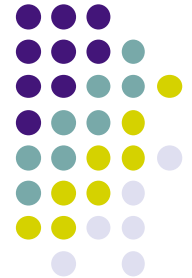
- We have introduced a simple and expressive paradigm for concurrent programming
- By design, it has no observable nondeterminism (no race conditions)
- It is based on two simple ideas
  - Synchronization of single-assignment variables on binding
  - Threads, a sequence of executing instructions
- We can build multi-agent programs using streams (a list with unbound tail) and agents (a list function running in a thread)
  - Deterministic dataflow is a form of functional programming

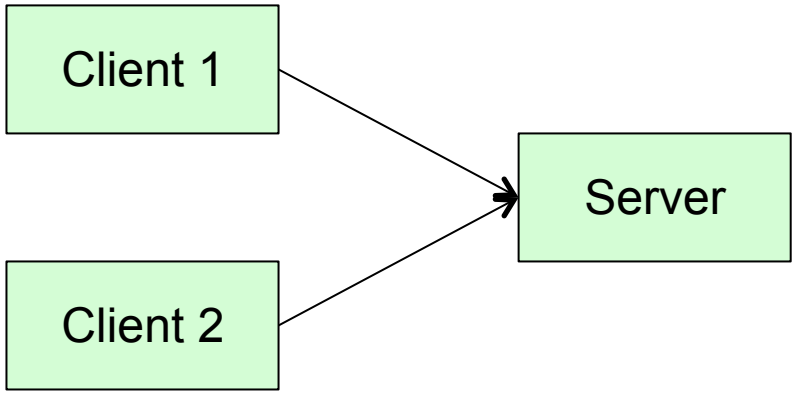
Concurrency *must* get simpler



- Parallel programming has finally arrived (a surprise to old timers like me!)
  - Multicore processors: dual and quad today, a dozen tomorrow, a hundred in a decade, soon most apps will do it
  - Distributed computing: data-intensive with tens of nodes today (NoSQL, MapReduce), hundreds and thousands tomorrow, most apps will do it
- Something fundamental will have to change
  - Sequential programming can't be the default (it's a centralized bottleneck)
  - Libraries can only hide so much (interface complexity, distribution structure)
- Concurrency will have to get a lot easier
  - Deterministic dataflow is functional programming!
  - It can be extended cleanly to distributed computing
    - Open network transparency
    - Modular fault tolerance
    - Large-scale distribution

But is determinism the right default?





A client/server can't be written in a deterministic paradigm!

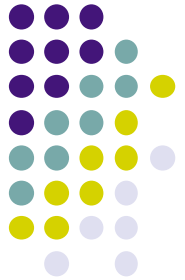
It's because the server must accept requests nondeterministically from the two clients

- Deterministic dataflow has strong limitations!
  - Any program that needs nondeterminism can't be written
  - Even a simple client/server can't be written
- But determinism has big advantages too
  - Race conditions are impossible by design
  - With determinism as default, we can reduce the need for nondeterminism (in the client/server, it's needed only at the point where the server accepts requests)
  - Any functional program can be made concurrent without changing the result

History of deterministic dataflow

- Deterministic concurrency has a long history that starts in 1974
  - Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, pp. 471-475, 1974. *Deterministic concurrency*.
  - Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pp. 993-998, 1977. *Lazy deterministic concurrency*.
- Why was it forgotten for so long?
  - Message passing and monitors arrived at about the same time:
    - Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 235-245, Aug. 1973.
    - Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549-557, Oct. 1974.
  - Actors and monitors express nondeterminism, so they are better. Right?
- Dataflow computing also has a long history that starts in 1974
  - Jack B. Dennis. First version of a data flow procedure language. *Springer Lecture Notes in Computer Science*, vol. 19, pp. 362-376, 1974.
  - Dataflow remained a fringe subject since it was always focused on parallel programming, which only became mainstream with the arrival of multicore processors in mainstream computing (e.g., IBM POWER4, the first dual-core processor, in 2001).

Next lesson



- General programming techniques for deterministic dataflow
  - « Concurrency for dummies »
- More sophisticated programming with deterministic dataflow
  - Higher-order programming and concurrent deployment
- Semantics of threads: how concurrency extends the abstract machine
  - A small extension to our abstract machine