A Concurrent-By-Default Programming Language

Sven Stork*† Paulo Marques* Jonathan Aldrich†

†Carnegie Mellon University *University of Coimbra

$(EC)^2$, July 15^{th}, 2011

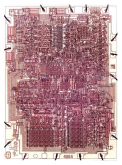
Information and Communication Technologies Institute
Carnegie Mellon | PORTUGAL
AN INTERNATIONAL PARTNERSHIP

Motivation

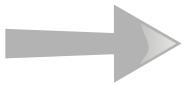
- The hardware changed
- The way we write software changed

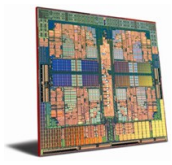
Motivation

- The hardware changed
  e.g., 2× Intel Xeon Processor X5660 $\Longrightarrow$ **24 cores**



4004



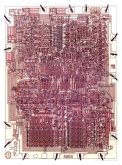


4-Core CPU

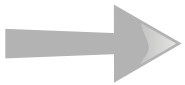
- The way we write software changed

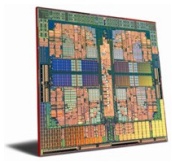
Motivation

- The hardware changed
  e.g., 2× Intel Xeon Processor X5660 $\Longrightarrow$ **24 cores**



4004





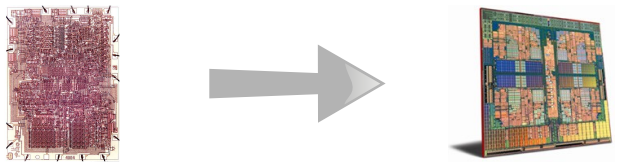
4-Core CPU

- The way we write software changed

programs

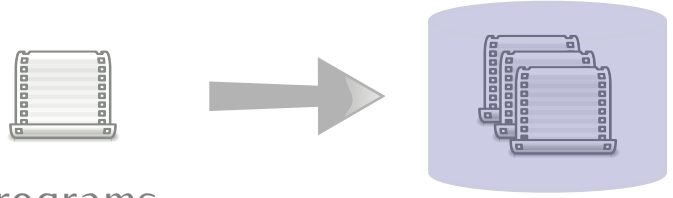
Motivation

- The hardware changed
  e.g., 2× Intel Xeon Processor X5660 $\Longrightarrow$ **24 cores**



4004 4-Core CPU

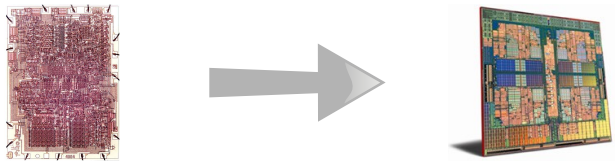
- The way we write software changed



programs libraries & frameworks

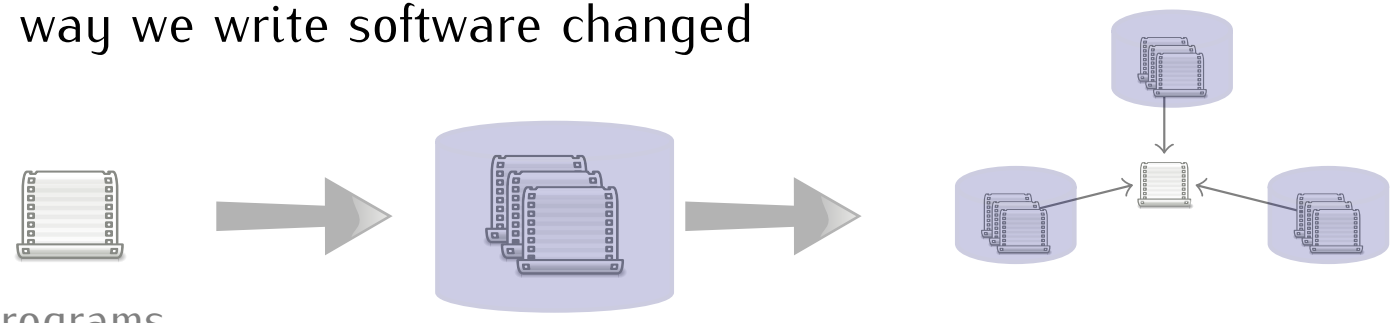
Motivation

- The hardware changed
  e.g., 2× Intel Xeon Processor X5660 $\Longrightarrow$ **24 cores**



4004 4-Core CPU

- The way we write software changed



programs libraries & frameworks glue-code programs

Wishlist for a possible solution

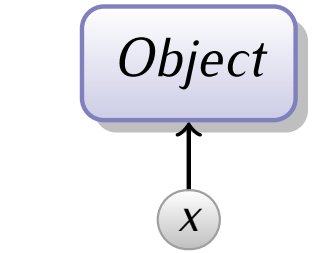
- Needs to be **composable**
- Needs to be **efficient** [1,2]
- composition needs to be **correct**
  (e.g., absence of race conditions [1,2])
- composition must **scale** well
- Handle concurrency/parallelism **automatically**
  (analogous garbage collection, JIT)

[1] Threads Cannot be Implemented as a Library (Boehm)

[2] Memory models: a case for rethinking parallel languages and hardware (Adve & Boehm)

Our take on that solution

- Use **aliasing information (access permissions)** and **abstract data collections (data groups)** to infer valid and conflicting concurrent execution combinations.



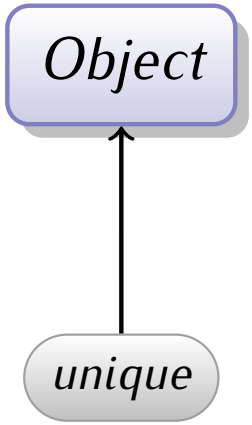
access permission

- Use **permission flow** to automatically parallelize execution in a dataflow fashion

What are access permissions?

- abstract capabilities associated with object references that encode
  - **access rights** (e.g., read/write)
  - **aliasing** information
- access permissions can be converted via **split** and **join** operations
- use **linear logic** and **fractions** to manage permissions
- extensively used for **verification** (e.g., concurrency, protocols, etc)

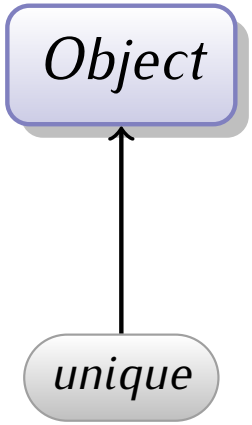
What permissions to we have?



Unique Permissions

- aliases = 1
- access = read/write
- only one reference to object
- exclusive access
- “thread local”
- no synchronization

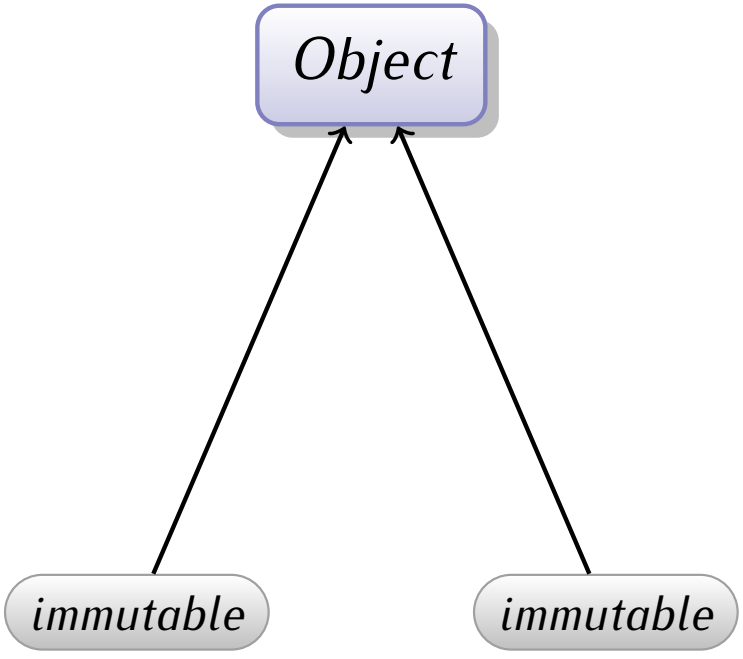
What permissions to we have?



Immutable Permissions

- aliases = N
- access = read
- all aliases are immutable
- “constant”
- no synchronization

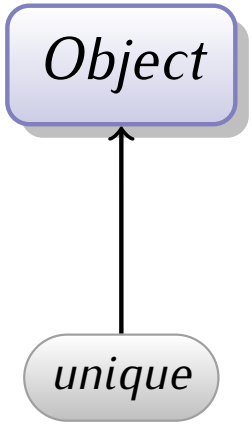
What permissions to we have?



Unique Permissions

- aliases = 1
- access = read/write
- only one reference to object
- exclusive access
- “thread local”
- no synchronization

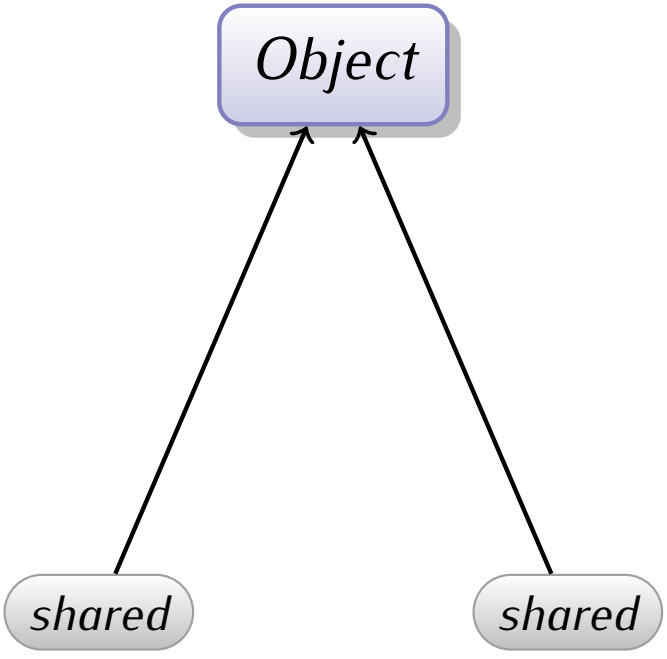
What permissions to we have?



Shared Permissions

- aliases = N
- access = read/write
- all aliases are shared
- synchronization

What permissions to we have?



Unique Permissions

- aliases = 1
- access = read/write
- only one reference to object
- exclusive access
- “thread local”
- no synchronization

How to use permissions to automatically parallelize?

- infer **permission flow** based on lexical order
- **define** operations can run in parallel iff the **intersection** of their required permissions does **not contain unique permissions**.
- **deterministic** execution iff intersection contains only **immutable** permissions
- **non-deterministic** execution iff intersection contains only **shared** permissions

Example: Bank Transfer

Bank Transfer

```
public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
    withdraw(from, amount)
    deposit(to, amount);
}
```

Example: Bank Transfer

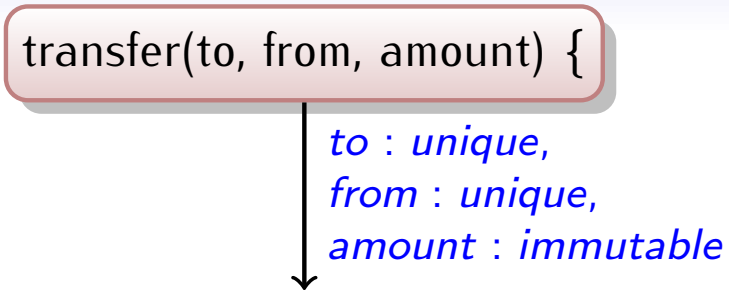
Bank Transfer

```
public void withdraw(unique Account account,
                     immutable Amount amount) {...}

public void deposit(unique Account account,
                    immutable Amount amount) {...}



public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
    withdraw(from, amount)
    deposit(to, amount);
}
```

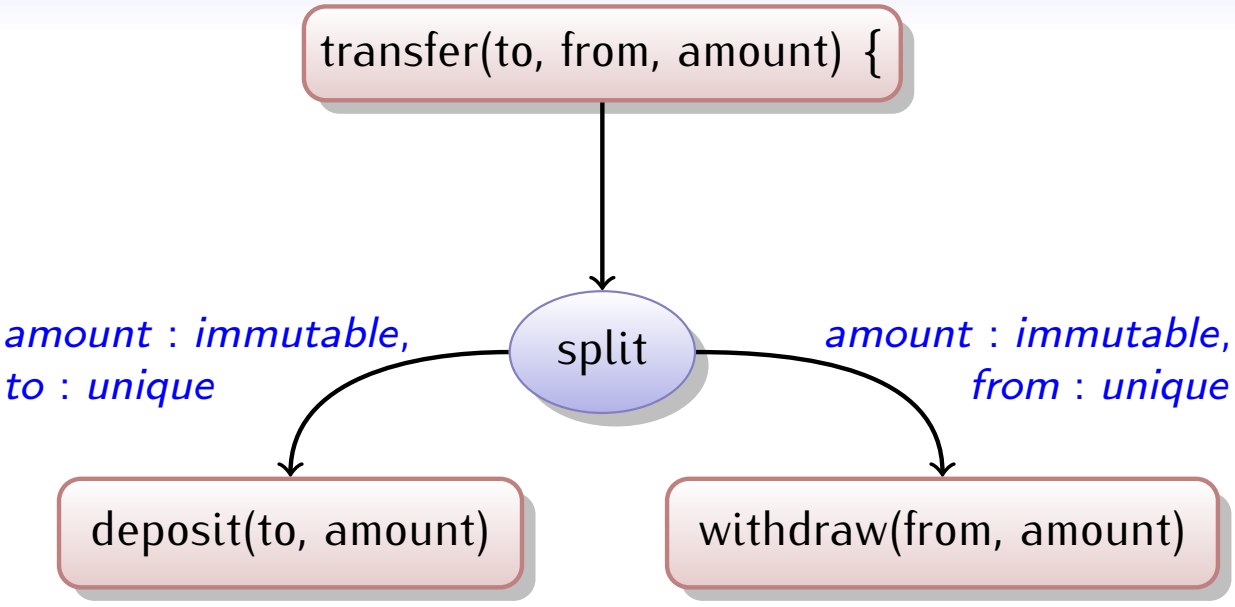
Example: Bank Transfer

```
transfer(to, from, amount) {
```

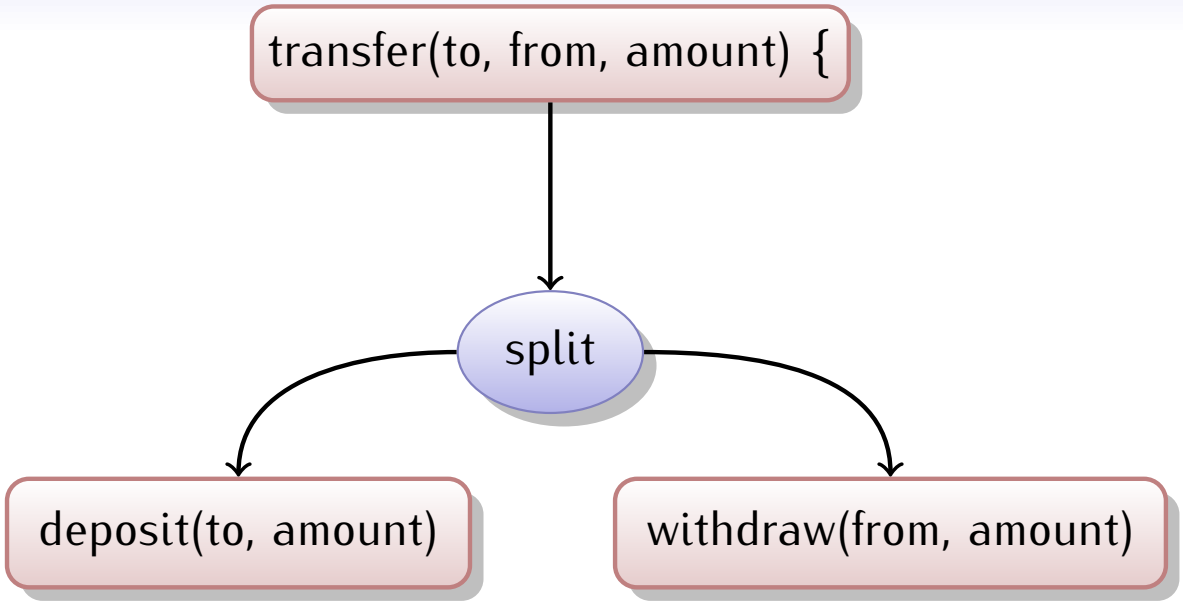
Example: Bank Transfer



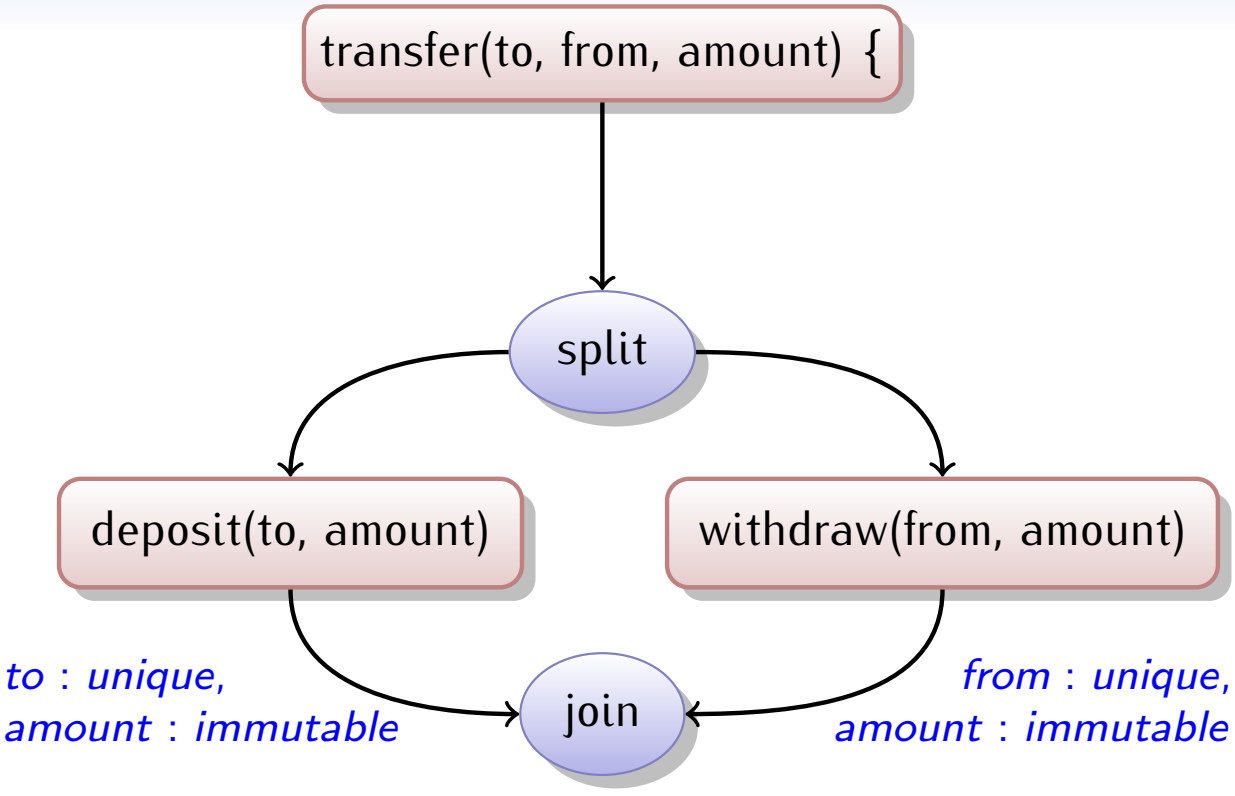
Example: Bank Transfer



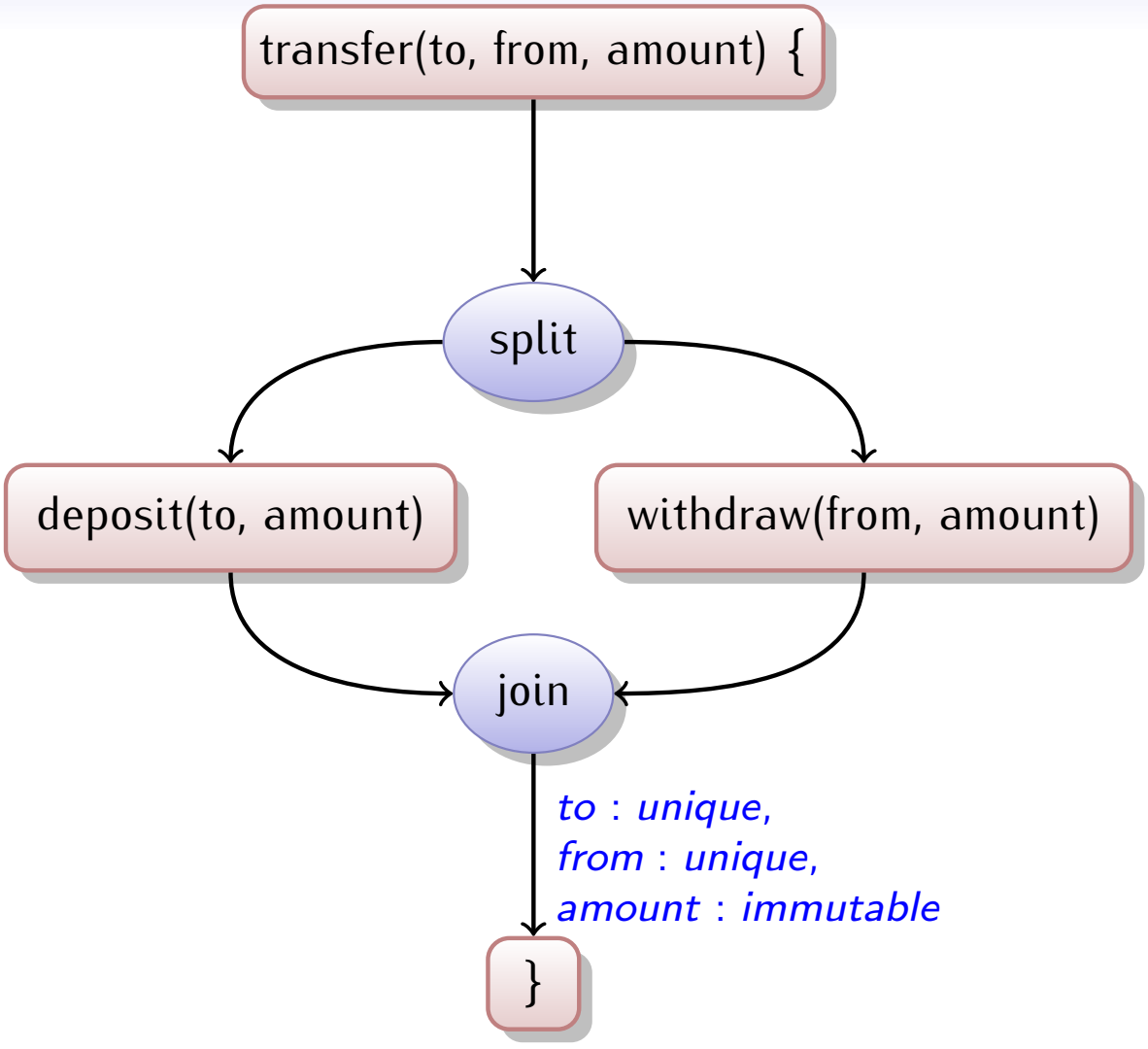
Example: Bank Transfer



Example: Bank Transfer



Example: Bank Transfer



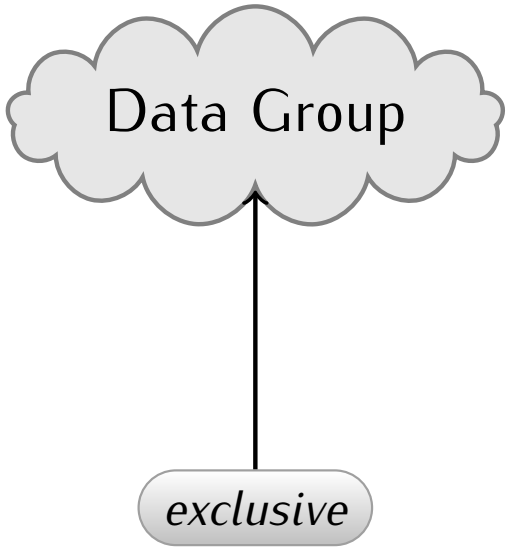
Shared data issues

- causes **non-determinism** but sometimes order matters
  - e.g., shared object that still needs to obey protocol
- **every access** to a shared object requires **synchronization**
  - sometimes shared is impossible to avoid
  - e.g., doubly linked list

Data Groups and Data Group Permissions

- bundle shared objects into **data groups**
  - abstract collection of objects
  - disjoint partitions of shared data
- introduce **data group permissions** for data groups
  - similar to access permissions for objects
  - manually split/joined by user via `split` block
  - allow the user to specify granularity

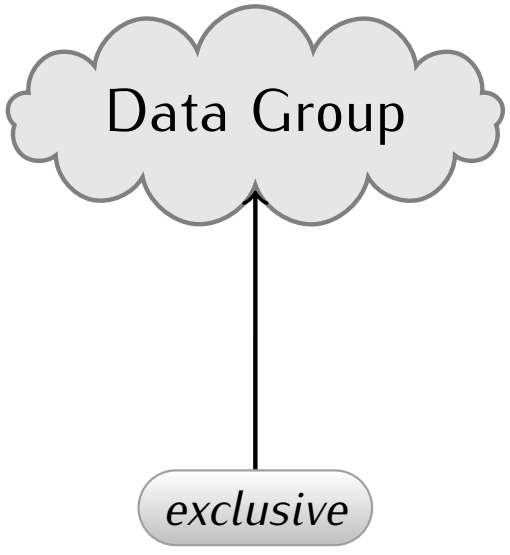
What permissions to we have?



Exclusive Permissions

- aliases = 1
- access = read/write
- only one reference to the data group
- exclusive access to data group
- “thread local”
- no synchronization

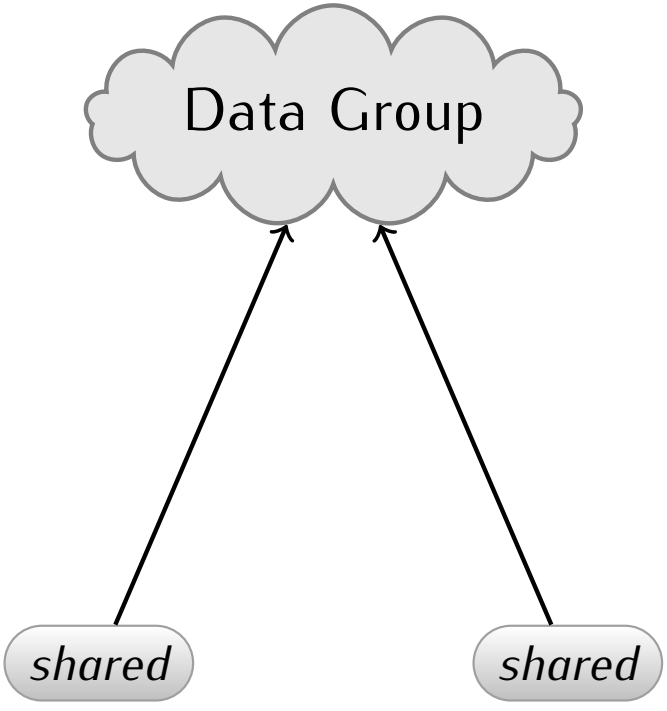
What permissions to we have?



Shared Permissions

- aliases = N
- access = none
- concurrent accesses to the same data group
- **manually** split by user
- no access to associated object
- requires synchronization

What permissions to we have?



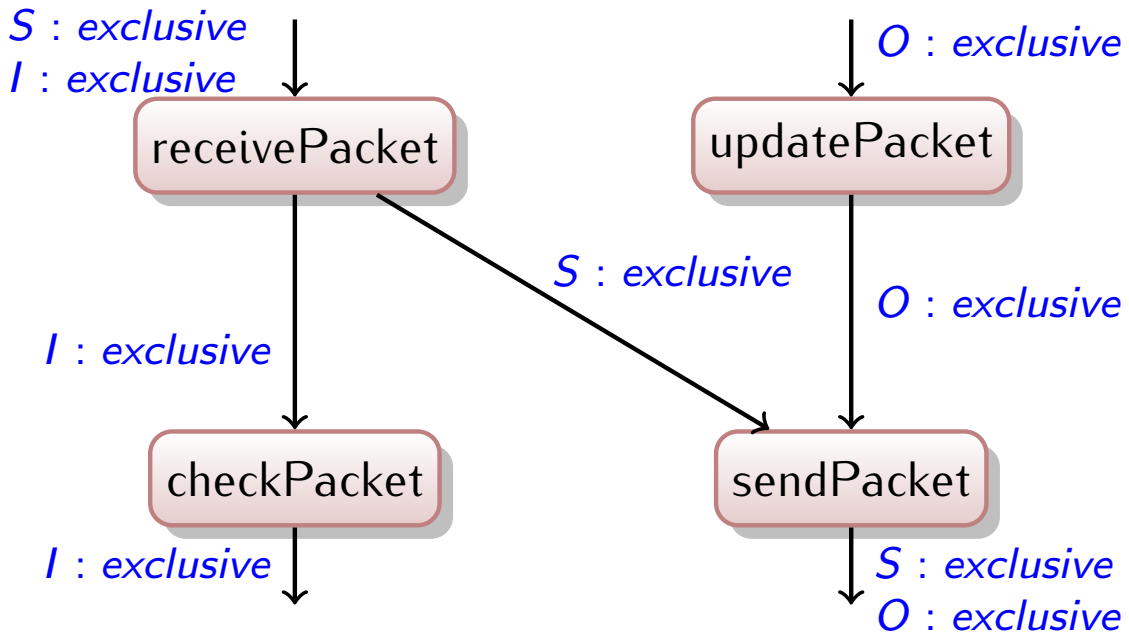
Atomic Permissions

- aliases = 1 atomic
  + (N-1) shared
- access = read/write
- protected access to shared data group objects
- "convert" shared permission to atomic permission via `atomic` block
- already synchronized

Example: Data Groups

```
void exchange⟨exclusive S,
              exclusive I,
              exclusive O⟩(shared⟨S⟩ Socket s,
                           shared⟨I⟩ Packet inp,
                           shared⟨O⟩ Packet outp) {
    receivePacket⟨S, I⟩(s, inp);
    checkPacket⟨I⟩(inp);
    updatePacket⟨O⟩(outp);
    sendPacket⟨S, O⟩(s, outp);
}
```

Example: Permission Flow of Data Groups



Conclusion

- **experiment** to see how far we can **push the envelop**
- try address the **concurrency** and **software engineering** issues
- **concurrent-by-default** approach based on **access permissions** and **data groups**

Thanks for the Attention!

Questions?

Possible problems

- **granularity** and **overhead**
  - use mixture between static and dynamic approach
- **annotation overhead**
  - implementing ÆMINIUM in Plaid which has permissions build-in
- how to deal with **legacy code**
  - provide external descriptions / wrapper libraries
- how to provide useful **feedback to the user** (e.g., visualization, debugging, etc)

(Some) Related Work

DPJ	fork/join approach, but lacks data flow and object granularity
Clairk et al.	parallel for-loops and dataflow approach for loop bodies, but only deterministic parallelism
NESL, ZPL	data parallelism only, only deterministic
Fortress	parallel for-loops and tuple-evaluation, but no checks
Cilk	explicit fork/join without checking

What does the name Æminium come from?

- Æminium was the **ancient roman city** on which **Coimbra** was established.