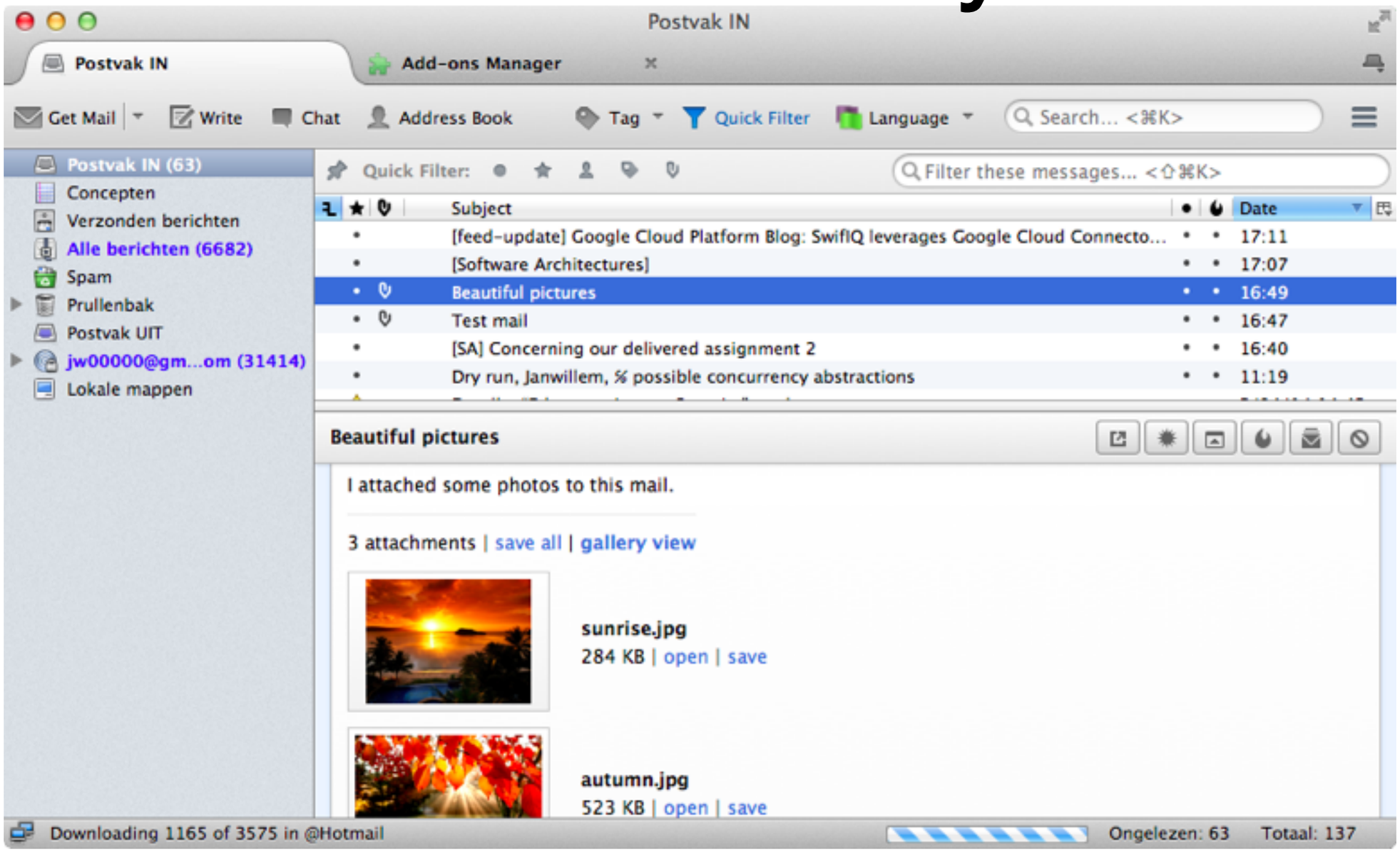
Towards Composable Concurrency Abstractions

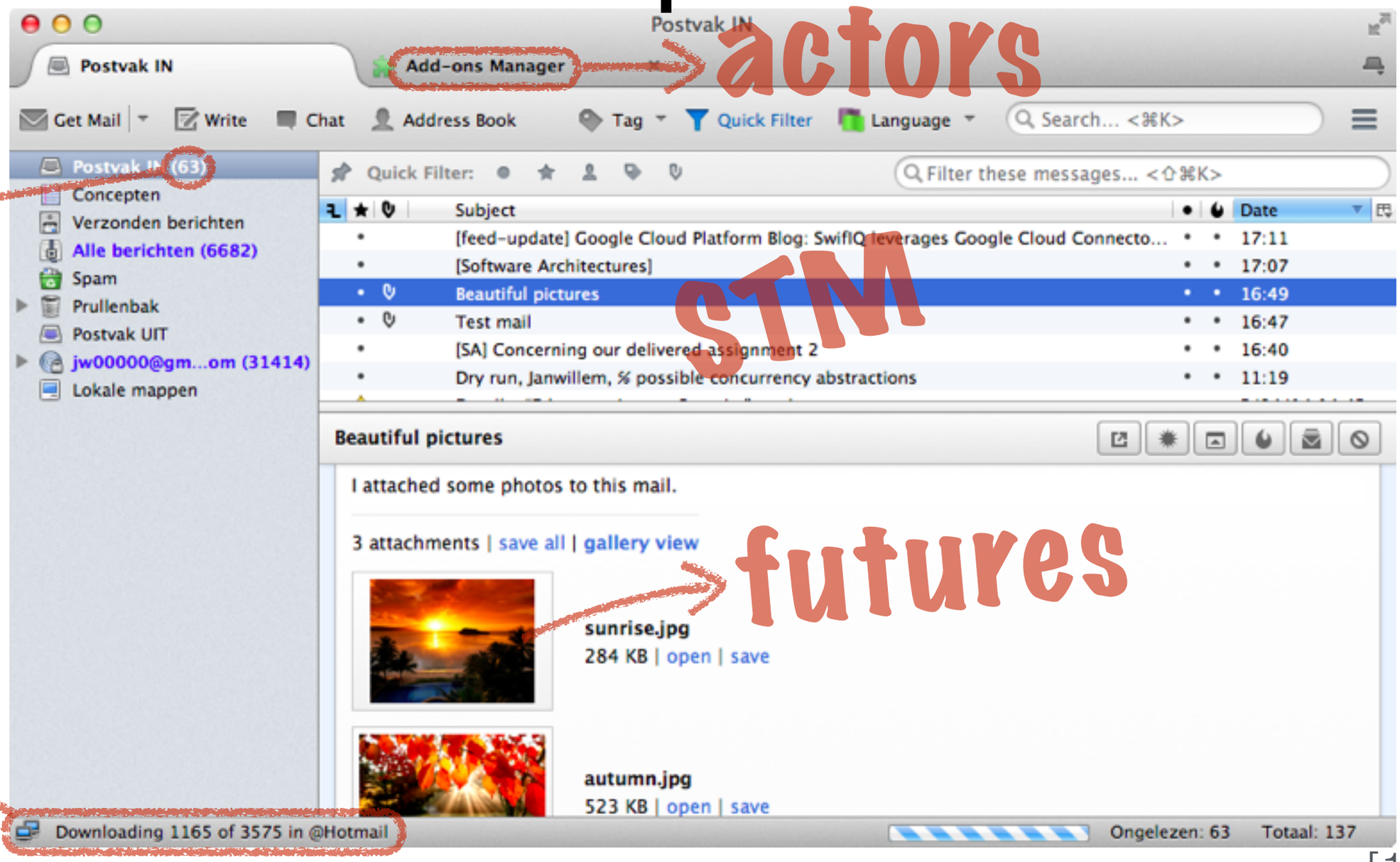
Janwillem Swalens
Stefan Marr
Joeri De Koster
Tom Van Cutsem

Software Languages Lab
Vrije Universiteit Brussel

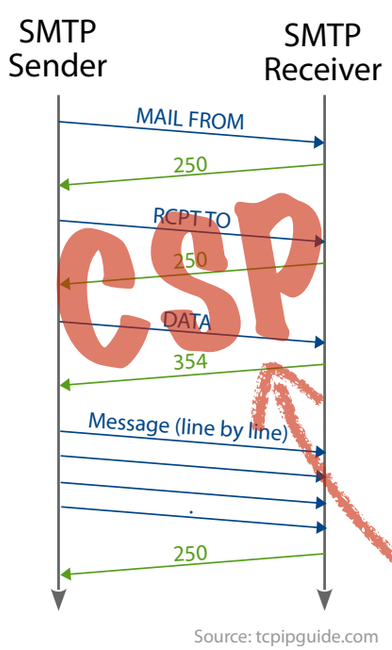
Complex GUI applications need concurrency



Different concurrency models for different requirements



atomic integer



Concurrency models are combined[1]

[1] Samira Tasharofi, Peter Dinges, and Ralph Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In Proc. of ECOOP'13, Montpellier, France, 2013.

Can concurrency models be **combined** correctly?

Are they **composable**?

“composable”?

Criteria:[2]

- **Safety**: partial correctness
  correct input → ~~incorrect output~~
- **Liveness**: termination
  correct input terminates

correct input
↓ eventually
correct output

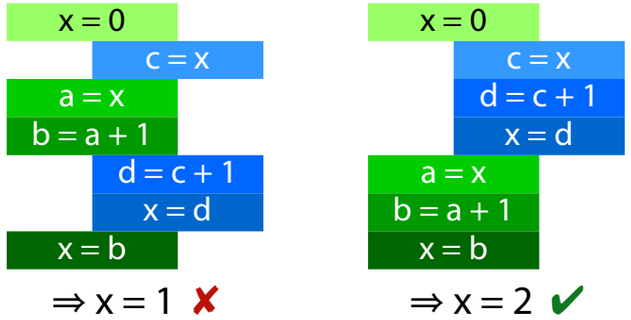
Two models are **composable** if combining them cannot produce *new* safety or liveness issues.

[2] Leslie Lamport. Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering, SE-3(2):125–143, 1977.

Safety and liveness issues

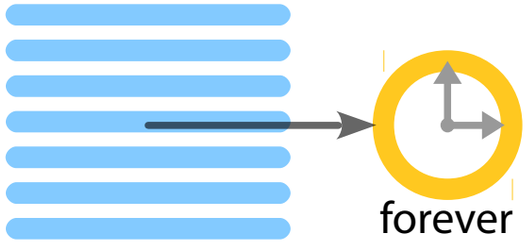
Safety:

→ **Race conditions**: incorrect results caused by bad interleavings



Liveness:

→ **Deadlocks**: introduced by blocking operations ⊗



→ **Livelocks**: code is re-executed under a certain condition ↻

Concurrency models

- Atomics
- Actors & Agents
- Software Transactional Memory (STM)
- Futures & Promises
- Communicating Sequential Processes (CSP)

→ Clojure

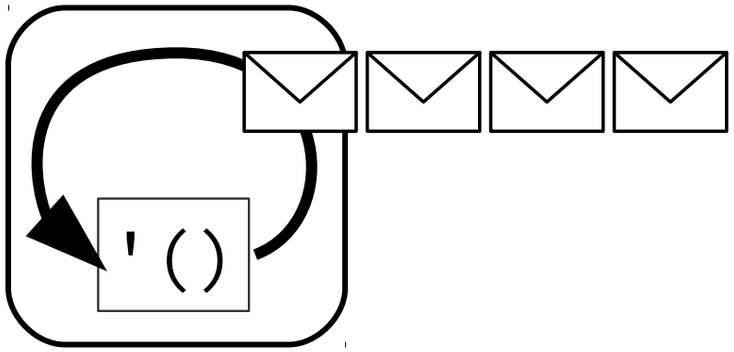
Atomic variables

```
(def unread-mails (atom 15))
(println (deref unread-mails))
(reset! unread-mails 0)
(swap! unread-mails inc)
```

	atoms
create	`atom`
read	`deref`
write	`reset!`
	`swap!` ↺

Agents

```
(def notifications (agent '()))
(println (deref notifications))
(send notifications
    (fn [msgs] (cons "Hello!" msgs)))
(await notifications)
```



	atoms	agents
create	`atom`	`agent`
read	`deref`	`deref`
write	`reset!`	
	`swap!` ↺	`send`
other		`await` ⊗

Software Transactional Memory

```
(def mail (ref {:subject "Hi!"}))
(dosync
    (println (deref mail))
    (ref-set mail {:subject "Hello!"})
    (alter mail (fn [m] (assoc m :subject "Hey!"))))
```

	atoms	agents	STM
create	atom	agent	ref
read	deref	deref	deref
write	reset!		ref-set
	swap! ↺	send	alter
block			dosync ↺
other		await ⊗	

Safety issues

Safety

used in	atoms	agents	STM	futures promises	CSP
atoms	✘	✘	✘	✘	✘
agents	✔	✔	✔	✔	✔
STM	✘	✔	✔	✘	✘
futures promises	✔	✔	✔	✔	✔
CSP	✔	✔	✔	✔	✔

```
(def notifications (agent '()))
(def unread-mails (atom 0))
(swap! unread-mails
  (fn [n]
    (send notifications
      (fn [msgs] (cons "New mail!" msgs)))
    (inc n)))
```

```
(def notifications (agent '()))
(def mail (ref {:subject "Hi" :archived false}))
(dosync
  (ref-set mail (assoc @mail :archived true))
  (send notifications
    (fn [msgs] (cons (str "Archived mail "
                          (:subject @mail)) msgs))))
```

Outer model re-executes } ⇒ **bad interleavings**

Inner model irrevocable actions } ⇒ **unsafe**

Futures & Promises

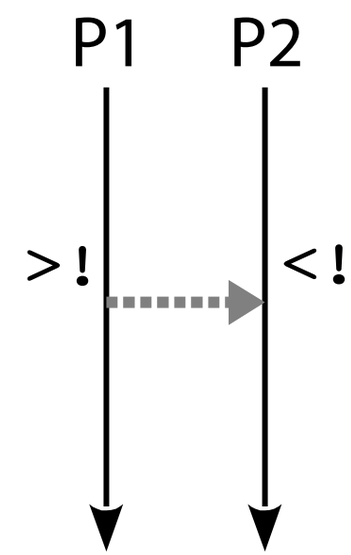
```
(def thumbnail (future (generate-thumbnail “attach.pdf”)))
(println (deref thumbnail))

(def p (promise))
(deliver p 2)
(println (deref p))
```

	atoms	agents	STM	futures	promises
create	atom	agent	ref	future	promise
read	deref	deref	deref	deref ⊗	deref ⊗
write	reset!		ref-set		deliver
	swap! ↺	send	alter		
block			dosync ↺		
other		await ⊗			

Communicating Sequential Processes

```
(def incoming-mails (chan))
(go
   (println (<! incoming-mails))
(go
   (>! incoming-mails {:subject “Hi!”}))
```



	atoms	agents	STM	futures	promises	CSP
create	atom	agent	ref	future	promise	chan
read	deref	deref	deref	deref ⊗	deref ⊗	<! ⊗
write	reset!		ref-set		deliver	>! ⊗
	swap! ↺	send	alter			
block			dosync ↺			go
other		await ⊗				

Liveness issues

Liveness

used in	atoms	agents	STM	futures promises	CSP
atoms	✔	✔	✔	✔	✘
agents	✔	✔	✔	✘	✘
STM	✔	✔	✔	✔	✘
futures promises	✔	✘	✔	✘	✘
CSP	✔	✘	✔	✔	✘

```
(def mails (ref '()))
(def incoming-mails (chan))
(dosync
  (ref-set mails
    (cons (<!! incoming-mails) @mails))
```

```
(def p (promise))
(def ag (agent 0))
```

Thread 1: Thread 2:

```
(deref p)          (deliver p 2)                ✔
```

```
(send ag                          (send ag                    ✘
  (fn [_] (deref p))                (fn [_] (deliver p 2)))
```

Inner model blocks
Outer model doesn't expect this } **⇒ possible deadlock**

Study Clojure

Safety

used / in	atoms	agents	STM	futures promises	CSP
atoms	✘	✘	✘	✘	✘
agents	✔	✔	✔	✔	✔
STM	✘	✔	✔	✘	✘
futures promises	✔	✔	✔	✔	✔
CSP	✔	✔	✔	✔	✔

Liveness

used / in	atoms	agents	STM	futures promises	CSP
atoms	✔	✔	✔	✔	✘
agents	✔	✔	✔	✘	✘
STM	✔	✔	✔	✔	✘
futures promises	✔	✘	✔	✘	✘
CSP	✔	✘	✔	✔	✘

Unsafe when:
Outer model re-executes
Inner model irrevocable actions

Possible deadlock when:
Inner model blocks
Outer model doesn't expect this

✘ proven by counter-example
✔ "proof" by argument

Solutions & ideas

1. Existing solutions

E.g. **`send`** to agent in **`dosync`** = delayed (Clojure)

2. Extend existing solutions

→ Delay deliver of promise in transaction

→ Disallow reading futures/promises *in* agent (↔ *before*)

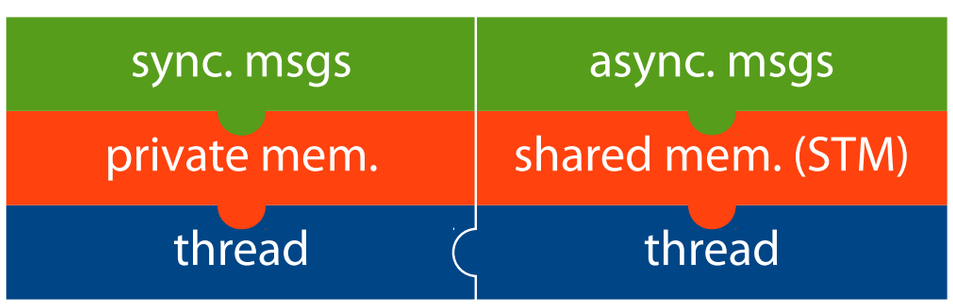
3. Future research

Building blocks

Future direction

Now:

Goal:

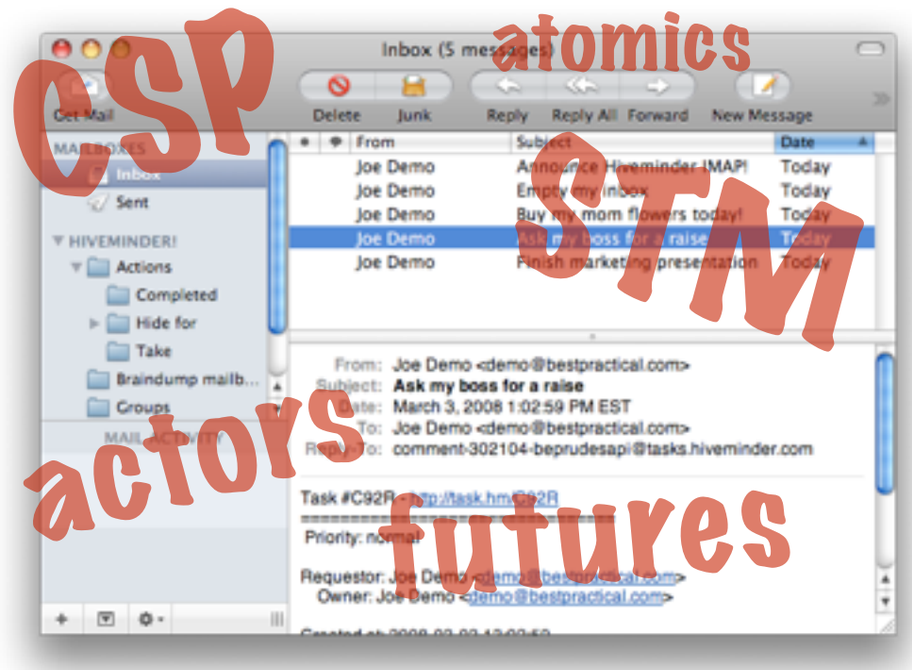


- safe, uniform, composition of components
- prevent unsafe combinations:
  - irrevocable actions in re-executing blocks (e.g. sync msgs in STM transaction)
  - blocking operations in blocks that guarantee progress (e.g. @future in agent)

Open questions & future work

- Proof ✔s
- Formal framework for all models?

Summary



Safety

	atoms	agents	refs	fut/prom	channel
atoms	✗	✗	✗	✗	✗
agents	✓	✓	✓	✓	✓
refs	✗	✓	✓	✗	✗
fut/prom	✓	✓	✓	✓	✓
channel	✓	✓	✓	✓	✓

Liveness

	atoms	agents	refs	fut/prom	channel
atoms	✓	✓	✓	✓	✗
agents	✓	✓	✓	✗	✗
refs	✓	✓	✓	✓	✗
fut/prom	✓	✗	✓	✗	✗
channel	✓	✗	✓	✓	✗