

6.1800 Spring 2025

Lecture #16: Atomicity, Isolation, Transactions

introducing abstractions to make fault-tolerance achievable

you have an exam tomorrow

there are a lot of things you can use to study, all on the website

- lecture outlines, slides
- recitation notes
- practice exams

the exam is open book but not open Internet. you will turn your network devices off during the exam. download everything you might need ahead of time.

you all can do well on this exam, get some sleep tonight

6.1800 in the news

How crawlers impact the operations of the Wikimedia projects

1 April 2025 by [Birgit Mueller, Wikimedia Foundation](#), [Chris Danis, Wikimedia Foundation](#) and [Giuseppe Lavagetto, Wikimedia Foundation](#)

Since January 2024, we have seen the bandwidth used for downloading multimedia content grow by 50%. This increase is not coming from human readers, but largely from automated programs that scrape the Wikimedia Commons image catalog of openly licensed images to feed images to AI models. Our infrastructure is built to sustain sudden traffic spikes from humans during high-interest events, but the amount of traffic generated by scraper bots is unprecedented and presents growing risks and costs.

The graph below shows that the base bandwidth demand for multimedia content has been growing steadily since early 2024 – and there’s no sign of this slowing down. This increase in baseline usage means that we have less room to accommodate exceptional events when a traffic surge might occur: a significant amount of our time and resources go into responding to non-human traffic.

6.1800 in the news

How crawlers impact the operations of the Wikimedia projects

1 April 2025 by [Birgit Mueller, Wikimedia Foundation](#), [Chris Danis, Wikimedia Foundation](#) and [Giuseppe Lavagetto, Wikimedia Foundation](#)

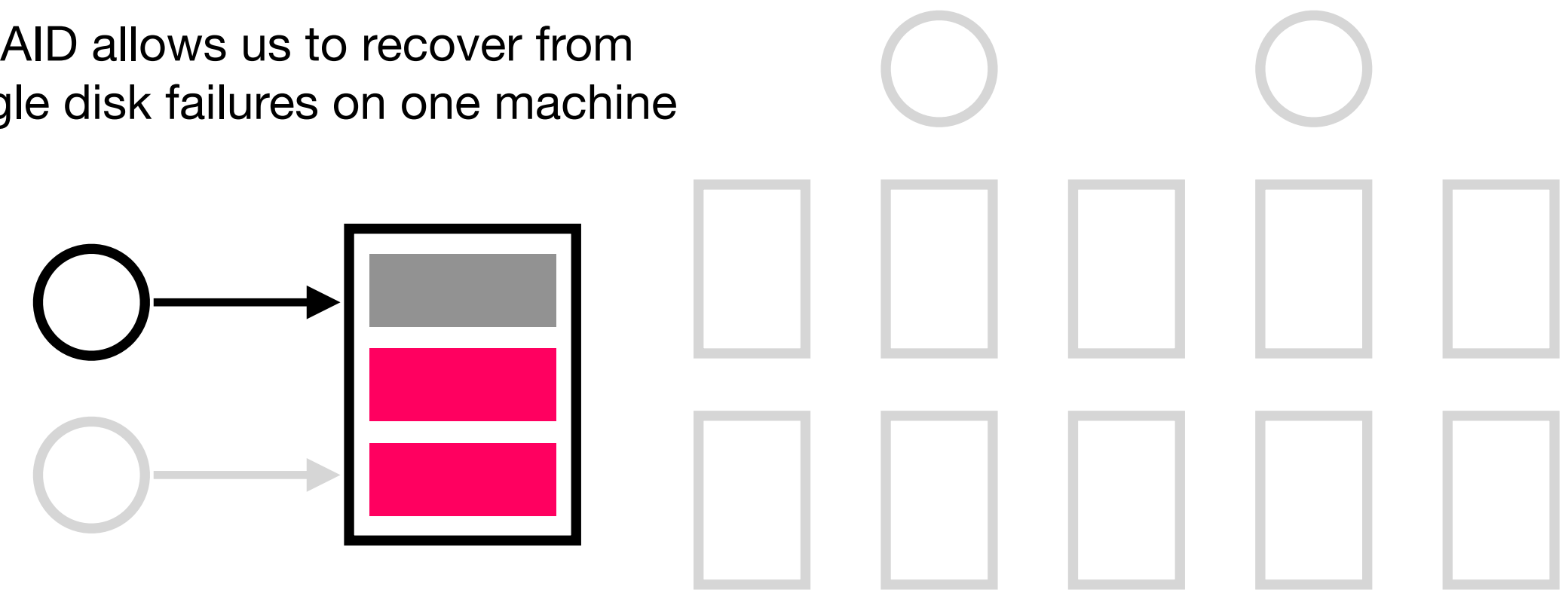
65% of our most expensive traffic comes from bots

The Wikimedia Foundation serves content to its users through a [global network of datacenters](#). This enables us to provide a faster, more seamless experience for readers around the world. When an article is requested multiple times, we memorize – or cache – its content in the datacenter closest to the user. If an article hasn’t been requested in a while, its content needs to be served from the core data center. The request then “travels” all the way from the user’s location to the core datacenter, looks up the requested page and serves it back to the user, while also caching it in the regional datacenter for any subsequent user.

While human readers tend to focus on specific – often similar – topics, crawler bots tend to “bulk read” larger numbers of pages and visit also the less popular pages. This means these types of requests are more likely to get forwarded to the core datacenter, which makes it much more expensive in terms of consumption of our resources.

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high

RAID allows us to recover from single disk failures on one machine



the high-level process of dealing with failures is to identify the faults, detect/contain the faults, and handle the faults. in lecture, we will build a **set of abstractions** to make that process more manageable

```
transfer (bank, account_a, account_b, amount):  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount ← crash! 💥
```

problem: `account_a` lost `amount` dollars, but
`account_b` didn't gain `amount` dollars

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

```
transfer (bank, account_a, account_b, amount):  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount ← crash! 💥
```

solution: make this action **atomic**. ensure that the system completes both steps or neither step.

current quest: update the bank transfer code to make this action *atomic*

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

atomicity

```
transfer (bank_file, account_a, account_b, amount):  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount ← crash! 💣  
    write_accounts(bank_file)
```

if the system crashes here, upon recovery, it will
appear as if the transfer didn't happen at all
because we didn't make any updates to **bank_file**

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

current quest: update the bank transfer code to make this action *atomic*

idea: write to a file so that a crash in between lines 2 and 3 has no effect


```
transfer (bank_file, account_a, account_b, amount):
```

```
    bank = read_accounts(bank_file)
```

```
    bank[account_a] = bank[account_a] - amount
```

```
    bank[account_b] = bank[account_b] + amount
```

```
    write_accounts(bank_file) ← crash! 💣
```

problem: a crash during `write_accounts()`
leaves `bank_file` in an intermediate state

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

current quest: update the bank transfer code to make this action *atomic*

idea: write to a file so that a crash in between lines 2 and 3 has no effect

```
transfer (bank_file, account_a, account_b, amount):  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount  
    write_accounts(tmp_file) ← crash! 💥  
    rename(tmp_file, bank_file)
```

if the system crashes here, upon recovery, it will appear as if the transfer didn't happen at all because we didn't make any updates to `bank_file`. we don't read from `tmp_file`, so it's okay if it was left in an intermediate state

current quest: update the bank transfer code to make this action *atomic*
idea: write to a temporary file so that a crash in between lines 2 and 3 has no effect, and neither does a crash during a write

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

```
transfer (bank_file, account_a, account_b, amount):  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount  
    write_accounts(tmp_file)  
    rename(tmp_file, bank_file) ← crash! 💥
```

problem: a crash during `rename()` potentially
leaves `bank_file` in an intermediate state

current quest: update the bank transfer code to make this action *atomic*

idea: write to a temporary file so that a crash in between lines 2 and 3
has no effect, and neither does a crash during a write

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

```
transfer (bank_file, account_a, account_b, amount):  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount  
    write_accounts(tmp_file)  
    rename(tmp_file, bank_file) ← crash! 💥
```

solution: make rename() atomic

**making rename() atomic is more feasible than
making write_accounts() atomic; we'll see why
as we go along**

current quest: update the bank transfer code to make this action *atomic*

idea: write to a temporary file so that a crash in between lines 2 and 3
has no effect, and neither does a crash during a write

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

relevant data structures

directory entries

filename “**bank_file**” -> inode **1**

filename “**tmp_file**” -> inode **2**

inode 1: // old data

data blocks: [...]

refcount: 1

inode 2: // new data

data blocks: [...]

refcount: 1

rename(tmp_file, orig_file):

// point orig_file's dirent at inode 2

// delete tmp_file's dirent

// remove refcount on inode 1

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

we're in an interlude, working on making rename atomic. this is the bank transfer code, which we'll eventually return to

transfer (bank_file, account_a, account_b, amount):

bank = read_accounts(bank_file)

bank[account_a] = bank[account_a] - amount

bank[account_b] = bank[account_b] + amount

write_accounts(tmp_file)

rename(tmp_file, bank_file)

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works

relevant data structures

directory entries

filename “**bank_file**” -> inode **2**

inode 1: // old data
data blocks: [...]
refcount: **0**

inode 2: // new data
data blocks: [...]
refcount: 1

```
rename(tmp_file, orig_file):  
    tmp_inode = lookup(tmp_file)    // = 2  
    orig_inode = lookup(orig_file)  // = 1
```

```
    orig_file dirent = tmp_inode  
    remove tmp_file dirent  
    decref(orig_inode)
```

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

we’re in an interlude, working on making rename atomic. this is the bank transfer code, which we’ll eventually return to

```
transfer (bank_file, account_a, account_b, amount):  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount  
    write_accounts(tmp_file)  
    rename(tmp_file, bank_file)
```

relevant data structures

directory entries

filename “**bank_file**” -> inode **1**

filename “**tmp_file**” -> inode **2**

inode 1: // old data

data blocks: [...]

refcount: 1

inode 2: // new data

data blocks: [...]

refcount: 1

rename(**tmp_file**, **orig_file**):

tmp_inode = lookup(**tmp_file**) // = 2

orig_inode = lookup(**orig_file**) // = 1

orig_file dirent = **tmp_inode** (here, or anywhere above this)

remove **tmp_file** dirent

decref(**orig_inode**)

← **crash!** 💥

it's as if rename didn't happen

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

we're in an interlude, working on making rename atomic. this is the bank transfer code, which we'll eventually return to

transfer (bank_file, account_a, account_b, amount):

bank = read_accounts(bank_file)

bank[account_a] = bank[account_a] - amount

bank[account_b] = bank[account_b] + amount

write_accounts(tmp_file)

rename(tmp_file, bank_file)

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works

relevant data structures

directory entries

filename “**bank_file**” -> inode **2**

filename “**tmp_file**” -> inode **2**

inode 1: // old data

data blocks: [...]

refcount: 1

inode 2: // new data

data blocks: [...]

refcount: 1

rename(**tmp_file**, **orig_file**):


tmp_inode = lookup(**tmp_file**) // = 2

orig_inode = lookup(**orig_file**) // = 1

orig_file dirent = **tmp_inode**

remove **tmp_file** dirent

decref(**orig_inode**)

← **crash!** 
(here, or anywhere after this)

**rename happened,
but refcounts might be wrong**

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

we’re in an interlude, working on making rename atomic. this is the bank transfer code, which we’ll eventually return to

transfer (bank_file, account_a, account_b, amount):

bank = read_accounts(bank_file)

bank[account_a] = bank[account_a] - amount

bank[account_b] = bank[account_b] + amount

write_accounts(tmp_file)

rename(tmp_file, bank_file)

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works

relevant data structures

directory entries

filename “**bank_file**” -> inode **?**

filename “**tmp_file**” -> inode **2**

inode 1: // old data

data blocks: [...]

refcount: 1

inode 2: // new data

data blocks: [...]

refcount: 1

rename(**tmp_file**, **orig_file**):

tmp_inode = lookup(**tmp_file**) // = 2

orig_inode = lookup(**orig_file**) // = 1

orig_file dirent = **tmp_inode** ← **crash!** 💣

remove **tmp_file** dirent

decref(**orig_inode**)

**crash *during* this line seems bad..
but is okay because single-sector writes
are themselves atomic**

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

we’re in an interlude, working on making rename atomic. this is the bank transfer code, which we’ll eventually return to

transfer (bank_file, account_a, account_b, amount):

bank = read_accounts(bank_file)

bank[account_a] = bank[account_a] - amount

bank[account_b] = bank[account_b] + amount

write_accounts(tmp_file)

rename(tmp_file, bank_file)

solution: recover from failure
(clean things up)

```
recover(disk):  
    for inode in disk.inodes:  
        inode.refcount = find_all_refs(disk.root_dir, inode)  
    if exists(tmp_file):  
        unlink(tmp_file)
```

**having a recovery process means that we don't
have to worry about getting everything
completely correct before the failure happens;
we have a chance to clean things up afterwards**

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

we're in an interlude, working on making rename atomic. this is the bank transfer code, which we'll eventually return to

```
transfer (bank_file, account_a, account_b, amount):  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount  
    write_accounts(tmp_file)  
    rename(tmp_file, bank_file)
```

atomicity

```
transfer (bank_file, account_a, account_b, amount):  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount  
    write_accounts(tmp_file)  
    rename(tmp_file, bank_file)
```

renaming the file — specifically modifying `bank_file`'s directory entry — is the commit point. if the system crashes before the commit point, it's as if the operation didn't happen; if it crashes after the commit point, the operation must complete. the commit point itself must also be atomic.

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

```
transfer (bank_file, account_a, account_b, amount):  
    acquire(lock)  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount  
    write_accounts(tmp_file)  
    rename(tmp_file, bank_file)  
    release(lock)
```

isolation deals with concurrency, and we've seen that.
couldn't we just put locks around everything?
isn't that what locks are *for*?

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

isolation

isolation refers to how and when the effects of one action (A1) are visible to another (A2). *in lecture*, we will aim to get a high level of isolation, where **A1 and A2 appear to have executed serially**, even if they are actually executed in parallel.


```
transfer (bank_file, account_a, account_b, amount):  
    acquire(lock)  
    bank = read_accounts(bank_file)  
    bank[account_a] = bank[account_a] - amount  
    bank[account_b] = bank[account_b] + amount  
    write_accounts(tmp_file)  
    rename(tmp_file, bank_file)  
    release(lock)
```

isolation deals with concurrency, and we've seen that.
couldn't we just put locks around everything?
isn't that what locks are *for*?

this particular strategy will perform poorly
would force a single transfer at a time

locks sometimes require global reasoning, which is messy
eventually, we'll incorporate locks, but in a systematic way

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

isolation

isolation refers to how and when the effects of one action (A1) are visible to another (A2). *in lecture*, we will aim to get a high level of isolation, where **A1 and A2 appear to have executed serially**, even if they are actually executed in parallel.

transactions provide atomicity and isolation

Transaction 1

```
begin
transfer(A, B, 20)
withdraw(B, 10)
end
```

Transaction 2

```
begin
transfer(B, C, 5)
deposit(A, 5)
end
```

**atomicity and isolation — and thus,
transactions — make it easier to reason
about failures (and concurrency)**

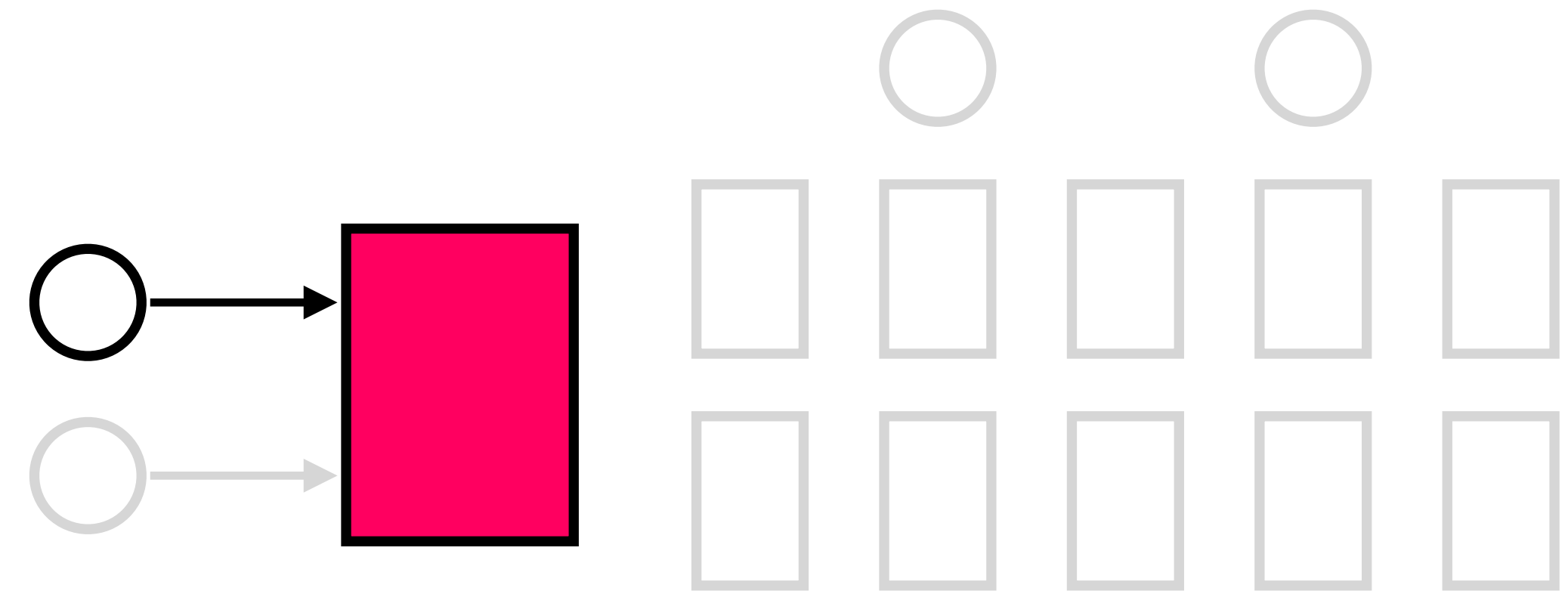
atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

isolation

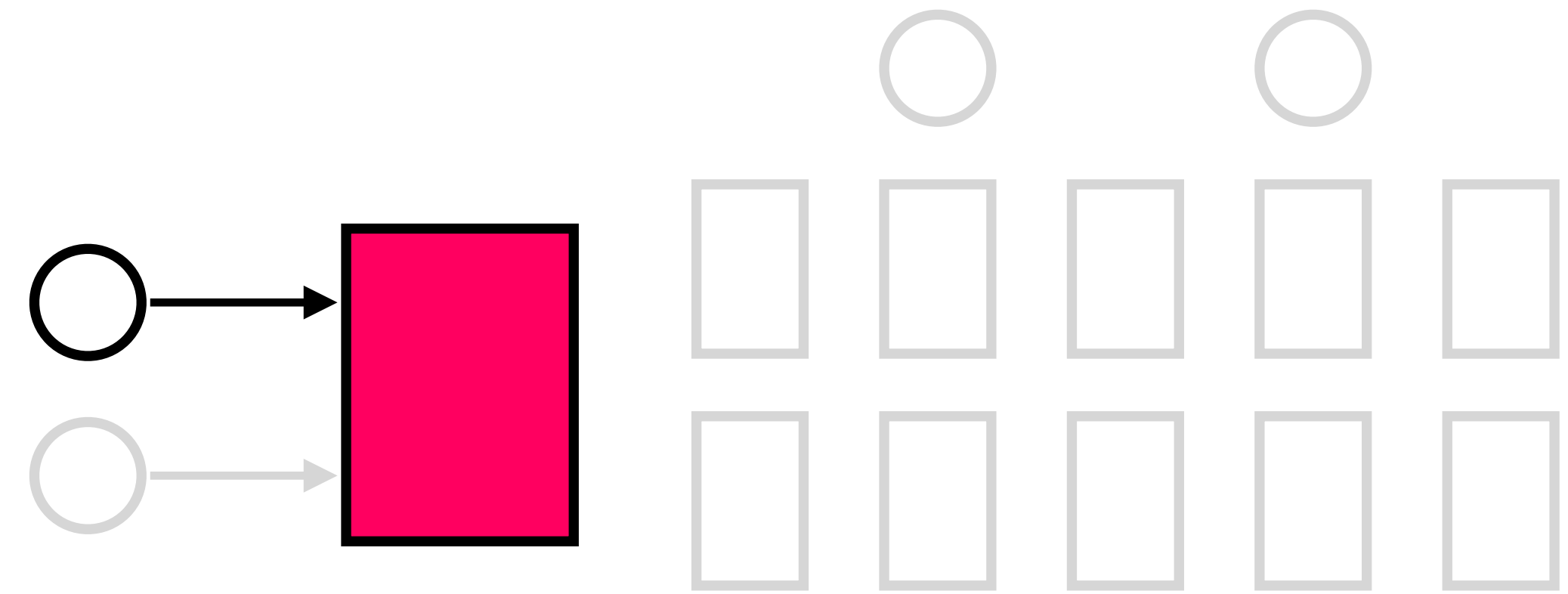
isolation refers to how and when the effects of one action (A1) are visible to another (A2). *in lecture*, we will aim to get a high level of isolation, where **A1 and A2 appear to have executed serially**, even if they are actually executed in parallel.

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



the high-level process of dealing with failures is to identify the faults, detect/contain the faults, and handle the faults. in lecture, we will build a **set of abstractions** to make that process more manageable

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

our job in lecture is to understand how a system *implements* transactions.
how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: we have this working for one user and one file via *shadow copies*, but they perform poorly

isolation: we don't really have this yet
coarse-grained locks perform poorly; fine-grained locks are difficult to reason about

transactions provide **atomicity** and **isolation**, both of which make it easier for us to reason about failures because we don't have to deal with intermediate states.

shadow copies are one way to achieve atomicity. they work in certain cases, but perform poorly: — requiring us to copy an entire file even for small changes — and don't allow for concurrency.

our main goal for the next few lectures is to *implement* transactions. how do we get the underlying system to provide atomicity and isolation so that this abstraction can exist?

we haven't covered how one would use shadow copies in general (e.g., outside of the world of banking). and we won't; next time, we'll work on a scheme that is superior in every way