Principles of Software Construction: Objects, Design, and Concurrency

Introduction, Overview, and Syllabus

Christian Kästner Bogdan Vasilescu

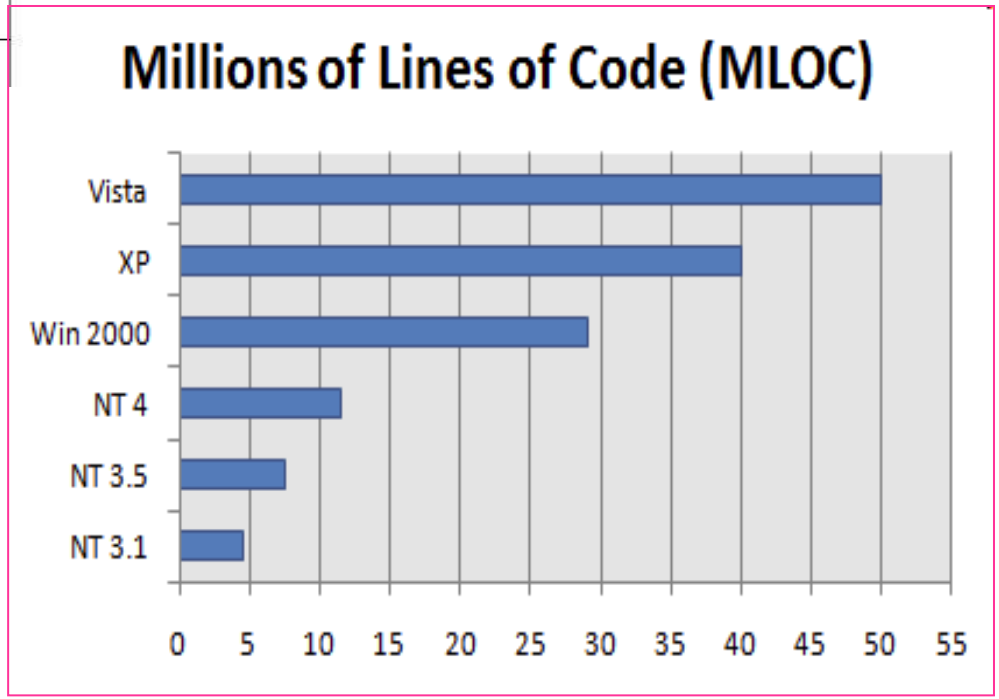
School of Computer Science
institute for SOFTWARE RESEARCH

Software is everywhere

institute for SOFTWARE RESEARCH

Growth of code and complexity

System	Year	% of Functions Performed in Software
F-4	1960	8
A-7	1964	10
F-111	1970	20
F-15	1975	35
F-16	1982	45
B-2	1990	65
F-22	2000	80



(informal reports)

COMMENTARY

Chris Murphy
Editor, InformationWeek

See more from this author

Tweet 164 Like 548 Share +1 21 Permalink

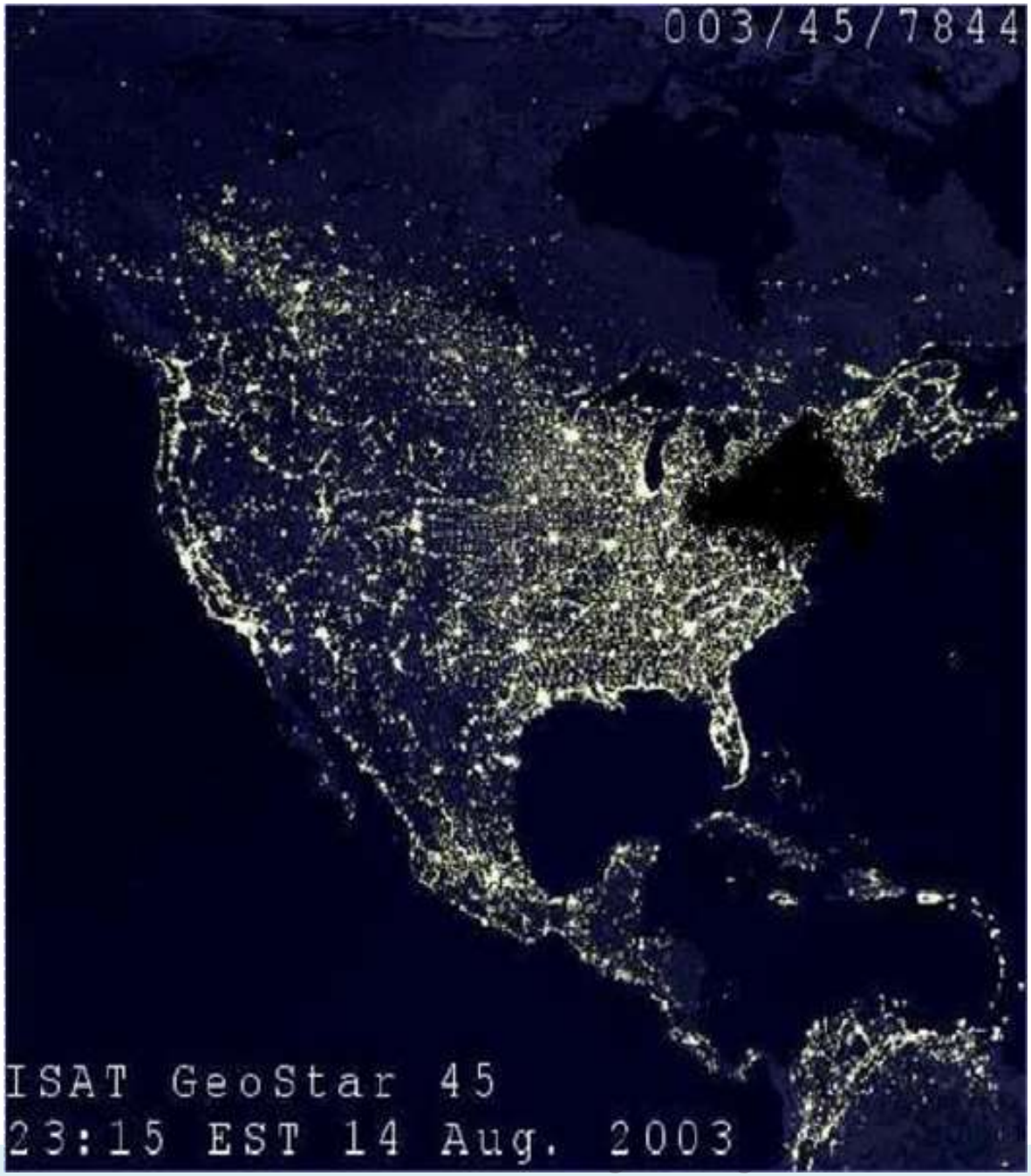
Why Ford Just Became A Software Company

Ford is upgrading its in-vehicle software on a huge scale, embracing all the customer expectations and headaches that come with the development lifecycle.

6 Comments | Chris Murphy | November 14, 2011 09:31 AM

Sometime early next year, Ford will mail USB sticks to about 250,000 owners of vehicles with its advanced touchscreen control panel. The stick will contain a major upgrade to the software for that screen. With it, Ford is breaking from a history as old as the auto industry, one in which the technology in a car essentially stayed unchanged from assembly line to junk yard.

Ford is significantly changing what a driver or passenger experiences in its cars years after they're built. And with it, Ford becomes a software company--with all the associated high customer expectations and headaches.


003/45/7844
ISAT GeoStar 45
23:15 EST 14 Aug. 2003

Principles of Software Construction: Objects, Design, and Concurrency

Introduction, Overview, and Syllabus

Christian Kästner Bogdan Vasilescu

primes

graph search

binary tree

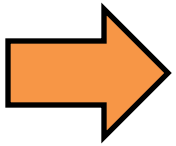
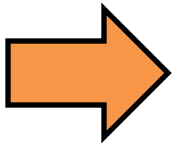
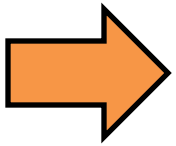
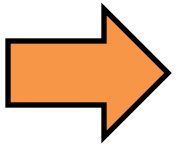
GCD

sorting

BDDs

institute for SOFTWARE RESEARCH

From Programs to Systems

Writing algorithms, data structures from scratch		Reuse of libraries, frameworks
Functions with inputs and outputs		Asynchronous and reactive designs
Sequential and local computation		Parallel and distributed computation
Full functional specifications		Partial, composable, targeted models

Our goal: understanding both the **building blocks** and also the **design principles** for construction of software systems at scale

Principles of Software Construction: Objects, Design, and Concurrency

Introduction, Overview, and Syllabus

Christian Kästner Bogdan Vasilescu

Objects in the real world

Object-oriented programming

- Programming based on structures that contain both data and methods

```
public class Bicycle {
  private int speed;
  private final Wheel frontWheel, rearWheel;
  private final Seat seat;
  …

  public Bicycle(…) { … }

  public void accelerate() {
    speed++;
  }

  public int speed() { return speed; }
}
```

Principles of Software Construction: Objects, Design, and Concurrency

Introduction, Overview, and Syllabus

Christian Kästner Bogdan Vasilescu

Semester overview

- Introduction to Java and O-O
- Introduction to **design**
  - **Design** goals, principles, patterns
- **Design**ing classes
  - **Design** for change
  - **Design** for reuse
- **Design**ing (sub)systems
  - **Design** for robustness
  - **Design** for change (cont.)
- **Design** case studies
- **Design** for large-scale reuse

- Explicit concurrency
- Distributed systems

- Crosscutting topics:
  - Modern development tools: IDEs, version control, build automation, continuous integration, static analysis
  - Modeling and specification, formal and informal
  - Functional correctness: Testing, static analysis, verification

Sorting with configurable order, variant A

```
void sort(int[] list, String order) {
    …
  boolean mustswap;
  if (order.equals("up")) {
    mustswap = list[i] < list[j];
  } else if (order.equals("down")) {
    mustswap = list[i] > list[j];
  }
  …
}
```

Sorting with configurable order, variant B

```
void sort(int[] list, Comparator cmp) {
    …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int I, int j) { return i>j; }}
```


(by the way, this design is called "strategy pattern")

institute for SOFTWARE RESEARCH

Tradeoffs?

```
void sort(int[] list, String order) {
   …
  boolean mustswap;
  if (order.equals("up")) {
    mustswap = list[i] < list[j];
  } else if (order.equals("down")) {
    mustswap = list[i] > list[j];
  }
  …
}
```

```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int I, int j) { return i>j; }}
```


institute for SOFTWARE RESEARCH

Sorting with a configurable order, version B'

```
interface Comparator {
  boolean compare(int i, int j);
}

final Comparator ASCENDING  = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
  …
  boolean mustSwap =
    cmp.compare(list[i], list[j]);
  …
}
```

it depends

it depends

(see context)

depends on what?
what are scenarios?
what are tradeoffs?

Software engineering is the branch of computer science that creates **practical, cost-effective solutions** to computing and information processing problems, preferably by applying scientific knowledge, developing software systems in the service of mankind.

Software Engineering for the 21st Century: A basis for rethinking the curriculum Manifesto, CMU-ISRI-05-108

"**Software engineering** is the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially by applying scientific knowledge, developing software systems in the service of mankind.
Software engineering entails making **decisions** under constraints of limited time, knowledge, and resources. […]

> Engineering quality resides in engineering judgment. […]
>
> Quality of the software product depends on the engineer's faithfulness to the engineered artifact. […]
>
> Engineering requires reconciling conflicting constraints. […]
>
> Engineering skills improve as a result of careful systematic reflection on experience. […]
>
> Costs and time constraints matter, not just capability. […]

Software Engineering for the 21st Century: A basis for rethinking the curriculum Manifesto, CMU-ISRI-05-108

Goal of software design

- Think before coding
- For each desired program behavior there are infinitely many programs
  - What are the differences between the variants?
  - Which variant should we choose?
  - How can we synthesize a variant with desired properties?
- Consider qualities
  - Maintainability, extensibility, performance, …
- Make explicit design decisions

Tradeoffs?

```
void sort(int[] list, String order) {
   …
  boolean mustswap;
  if (order.equals("up")) {
    mustswap = list[i] < list[j];
  } else if (order.equals("down")) {
    mustswap = list[i] > list[j];
  }
  …
}
```

```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int I, int j) { return i>j; }}
```

Preview: Design goals, principles, and patterns

- ***Design goals*** enable evaluation of designs
  - e.g. maintainability, reusability, scalability
- ***Design principles*** are heuristics that describe best practices
  - e.g. high correspondence to real-world concepts
- ***Design patterns*** codify repeated experiences, common solutions
  - e.g. template method pattern

institute for SOFTWARE RESEARCH

Software Engineering at CMU

- 15-214: “Code-level” design
  - extensibility, reuse, concurrency, functional correctness
- 15-313: “Human aspects” of software development
  - requirements, team work, scalability, security, scheduling, costs, risks, business models
- 15-413, 17-413 Practicum, Seminar, Internship
- Various master-level courses on requirements, architecture, software analysis, etc
- SE Minor: http://isri.cmu.edu/education/undergrad/

This is not a Java course

This is not a Java course

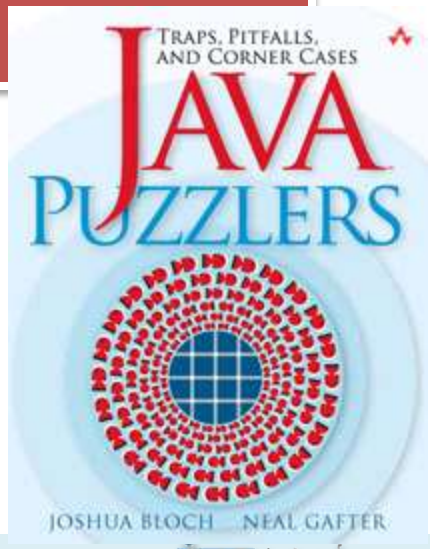
but you will write a lot of Java code

This is secretly a Java course

institute for SOFTWARE RESEARCH

```
int a = 010 + 3;
System.out.println("A" + a);
```

```
int a = 010 + 3;
System.out.println("A" + a);
```



Principles of Software Construction: Objects, Design, and Concurrency

Introduction, Overview, and Syllabus

Christian Kästner Bogdan Vasilescu

Summary: Course themes

- Object-oriented programming
- Code-level design
- Analysis and modeling
- Concurrency and distributed systems

COURSE ORGANIZATION

Course preconditions

- 15-122 or equivalent
  - Two semesters of programming
  - Knowledge of C-like languages
- 21-127 or equivalent
  - Familiarity with basic discrete math concepts
- Specifically:
  - Basic programming skills
  - Basic (formal) reasoning about programs
    - Pre/post conditions, invariants, formal verification
  - Basic algorithms and data structures
    - Lists, graphs, sorting, binary search, etc.

High-level learning goals

1. Ability to **design** medium-scale programs
   - Design goals (e.g., design for change, design for reuse)
   - Design principles (e.g., low coupling, explicit interfaces)
   - Design patterns (e.g., strategy pattern, decorator pattern), libraries, and frameworks
   - Evaluating trade-offs within a design space
   - Paradigms such as event-driven GUI programming
2. Understanding **object-oriented programming** concepts and how they support design decisions
   - Polymorphism, encapsulation, inheritance, object identity
3. Proficiency with basic **quality assurance** techniques for functional correctness
   - Unit testing
   - Static analysis
   - (Verification)
4. Fundamentals of **concurrency and distributed systems**
5. Practical skills
   - Ability to write medium-scale programs in Java
   - Ability to use modern development tools, including VCS, IDEs, debuggers, build and test automation, static analysis, …

(See course web site for a full list of learning goals)

Course staff

- Christian Kästner
  kaestner@cs.cmu.edu, Wean 5122
- Bogdan Vasilescu
  vasilescu@cmu.edu, Wean 5115
- Teaching assistants:
  Hubert, Zilei, Alvin, Evans, Avi, Jordan, Tianyu

Recitations are required

Course meetings

- Lectures: Tuesday and Thursday 3:00 – 4:20pm here :)

Recitation attendance is required

- Recitations: Wednesdays 9:30 - … - 3:20pm
  - Supplementary material, hands-on practice, feedback
  - Starting next week
- Office hours: see course web page
  - https://www.cs.cmu.edu/~ckaestne/15214/s2017/

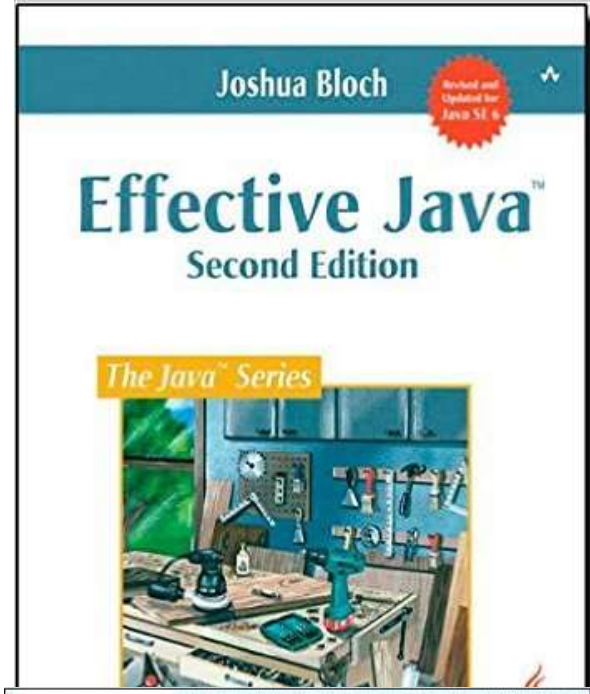
Course Infrastructure

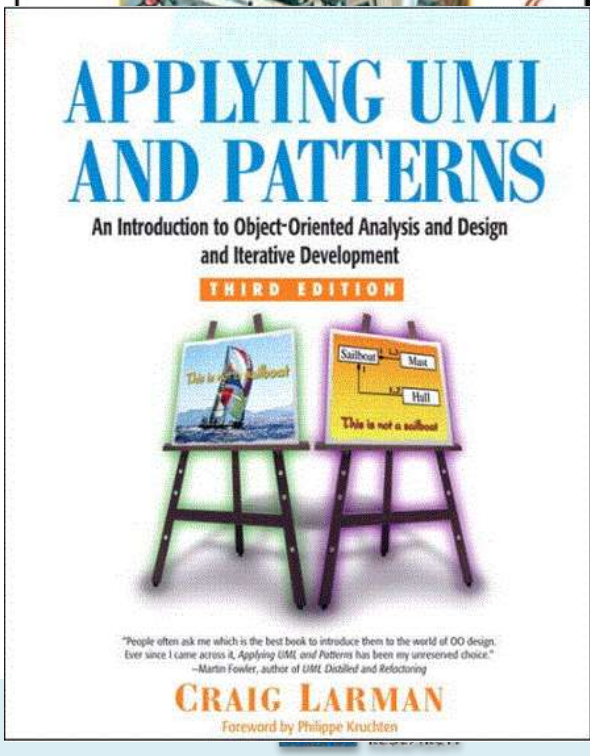
- Course website https://www.cs.cmu.edu/~ckaestne/15214/s2017/
  - Schedule, assignments, lecture slides, policy documents
- Tools
  - Git, Github: Assignment distribution, hand-in, and grades
  - Piazza: Discussion board
  - Eclipse, IntelliJ Idea, or similar: Recommended for developing code
  - Gradle, Travis-CI, Checkstyle, Findbugs: Practical development tools
- Assignments
  - Homework 1 available tomorrow morning
  - Ensure all tools are working together, Git, Java, Eclipse, Gradle, Checkstyle
  - Attend office hours in case of problems or ask on Piazza

Textbooks

- **Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd Edition. Prentice Hall. 2004. ISBN 0-13-148906-2**
- **Joshua Bloch. Effective Java, Second Edition. Addison-Wesley, ISBN 978-0321356680.**
- Selective other readings later in the semester
- Electronic version available through CMU library

- Regular reading assignments of chapters and online quizzes (Tuesdays)
- Additional (optional) readings listed on slides and web page





Approximate grading policy

- 50% assignments
- 20% midterms (2 x 10% each)
- 20% final exam
- 10% quizzes and participation

This course does not have a fixed letter grade policy; i.e., the final letter grades will not be A=90-100%, B=80-90%, etc.

Collaboration policy

- See course web page for details!
- We expect your work to be your own
- Do not release your solutions (not even after end of semester)
- Ask if you have any questions
- If you are feeling desperate, please reach out to us
  - Always turn in any work you've completed *before* the deadline
- We run cheating detection tools. Trust us, academic integrity meetings are painful for everybody

Late day policy

- See syllabus on course web page for details
- 2 possible late days per deadline (exceptions will be announced)
  - 5 total free late days for semester (+ separate 2 late days for assignments done in pairs)
  - 10% penalty per day after free late days are used
  - but we won't accept work 3 days late
- Extreme circumstances – talk to us

Principles of Software Construction: Objects, Design, and Concurrency

Introduction to Software Engineering tools

Christian Kästner **Bogdan Vasilescu**

School of Computer Science

institute for SOFTWARE RESEARCH

You will need for homework 1

- Java: more on Thursday

- Version control: Git
- Hosting: GitHub
- Build manager: Gradle
- Continuous integration service: Travis-CI

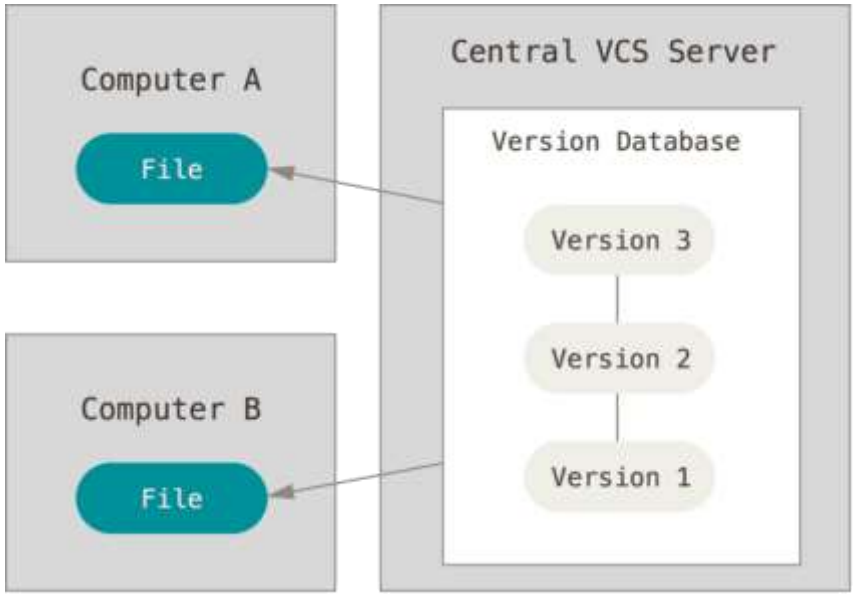
What is version control?

- System that records changes to a set of files over time
  - Revert files back to a previous state
  - Revert entire project back to a previous state
  - Compare changes over time
  - See who last modified something that might be causing a problem
- As opposed to:

hw1.java	hw1_v2.java	hw1_v3.java
hw1_final.java	hw1_final_new.java	…

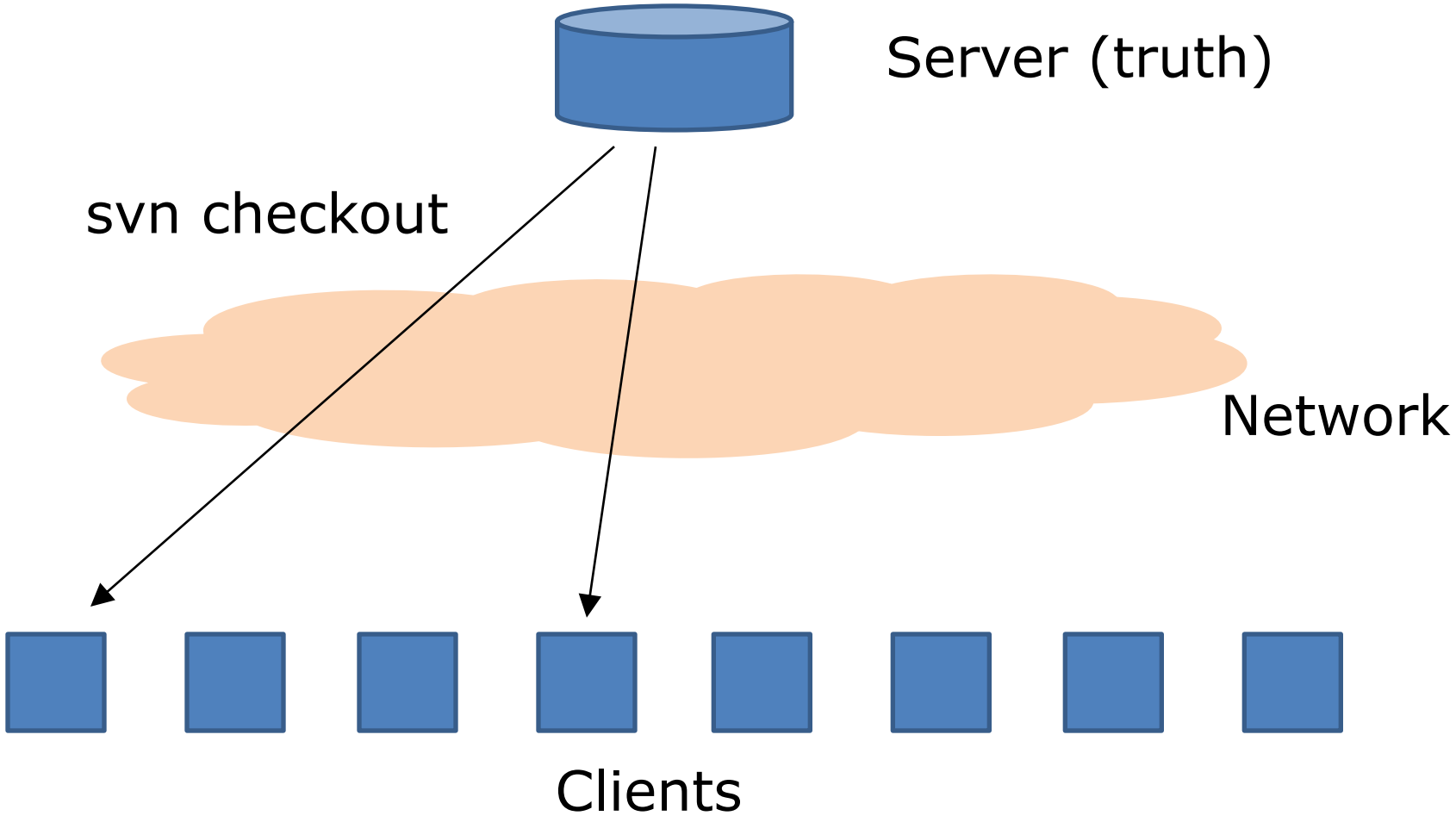
Centralized version control

- Single server that contains all the versioned files
- Clients check out/in files from that central place
- E.g., CVS, SVN (Subversion), and Perforce

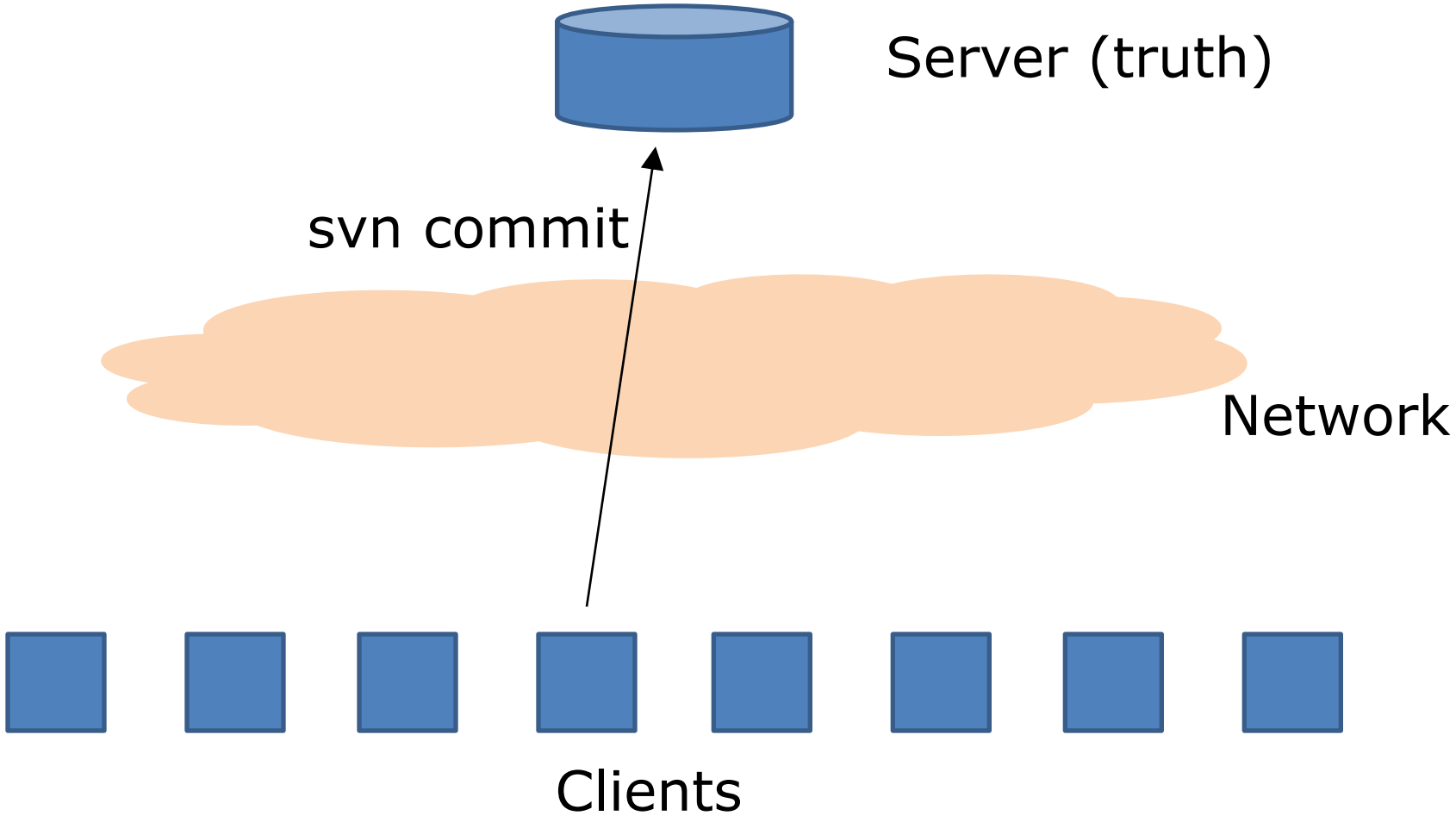


https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

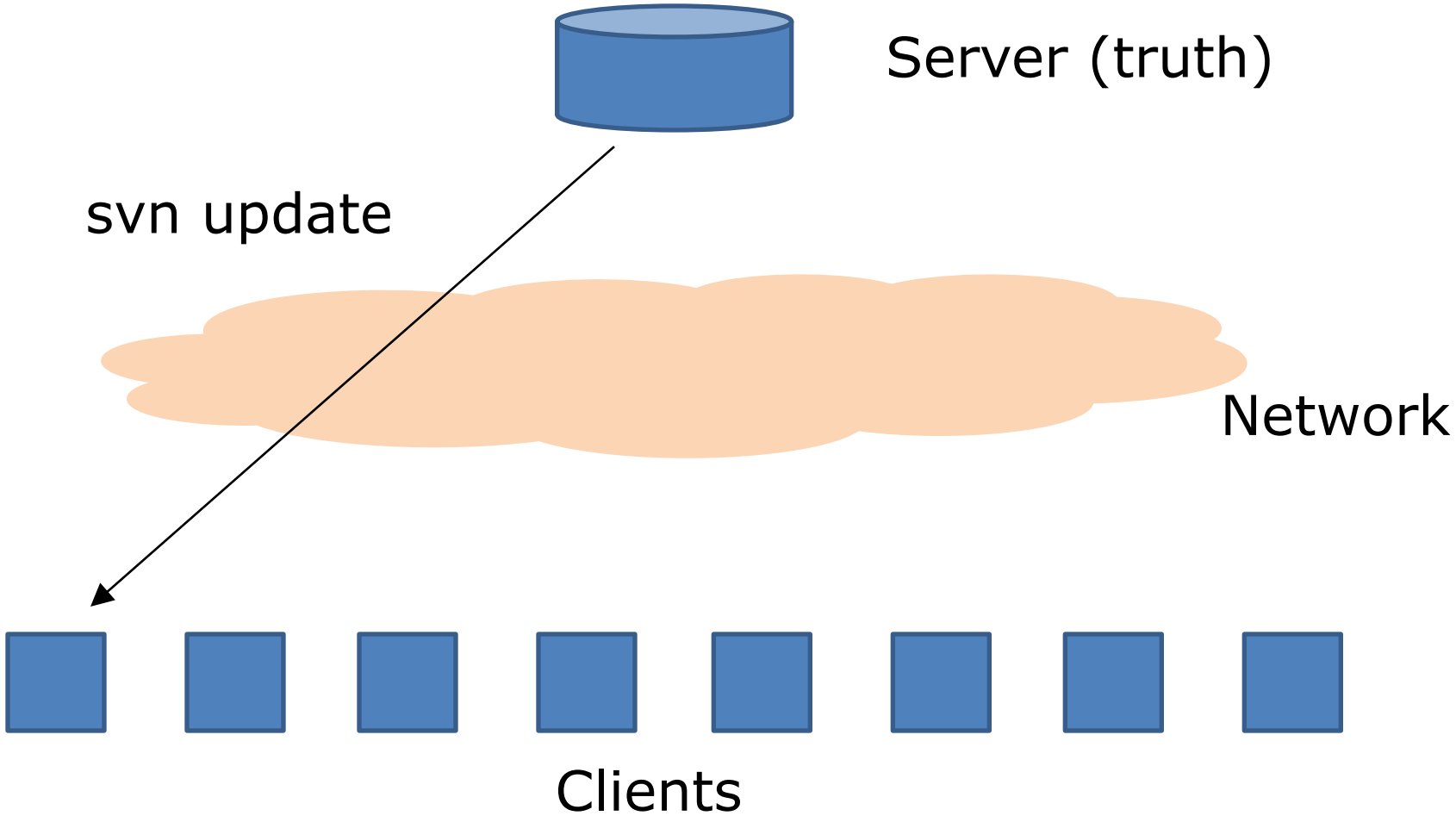
SVN



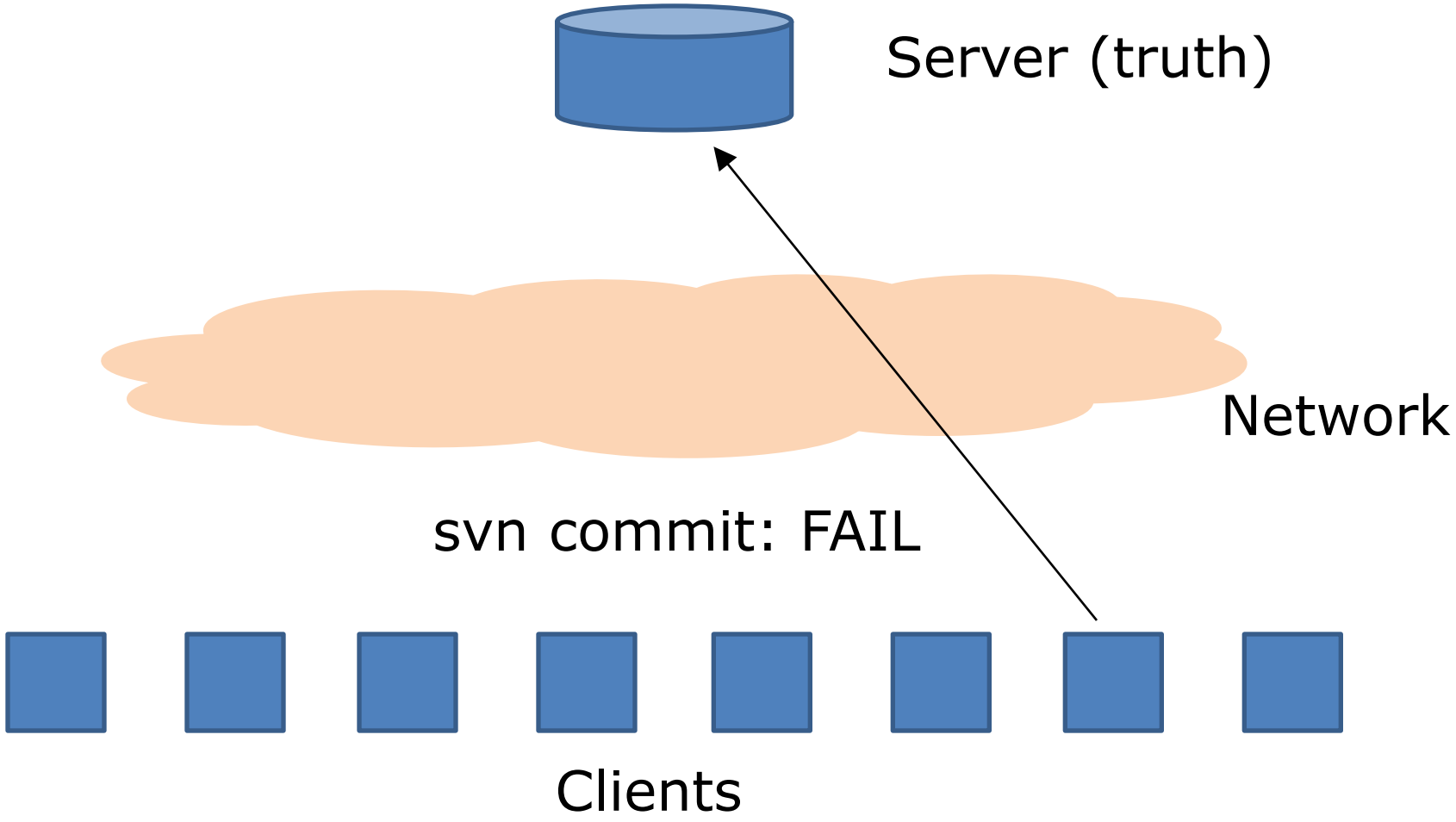
SVN



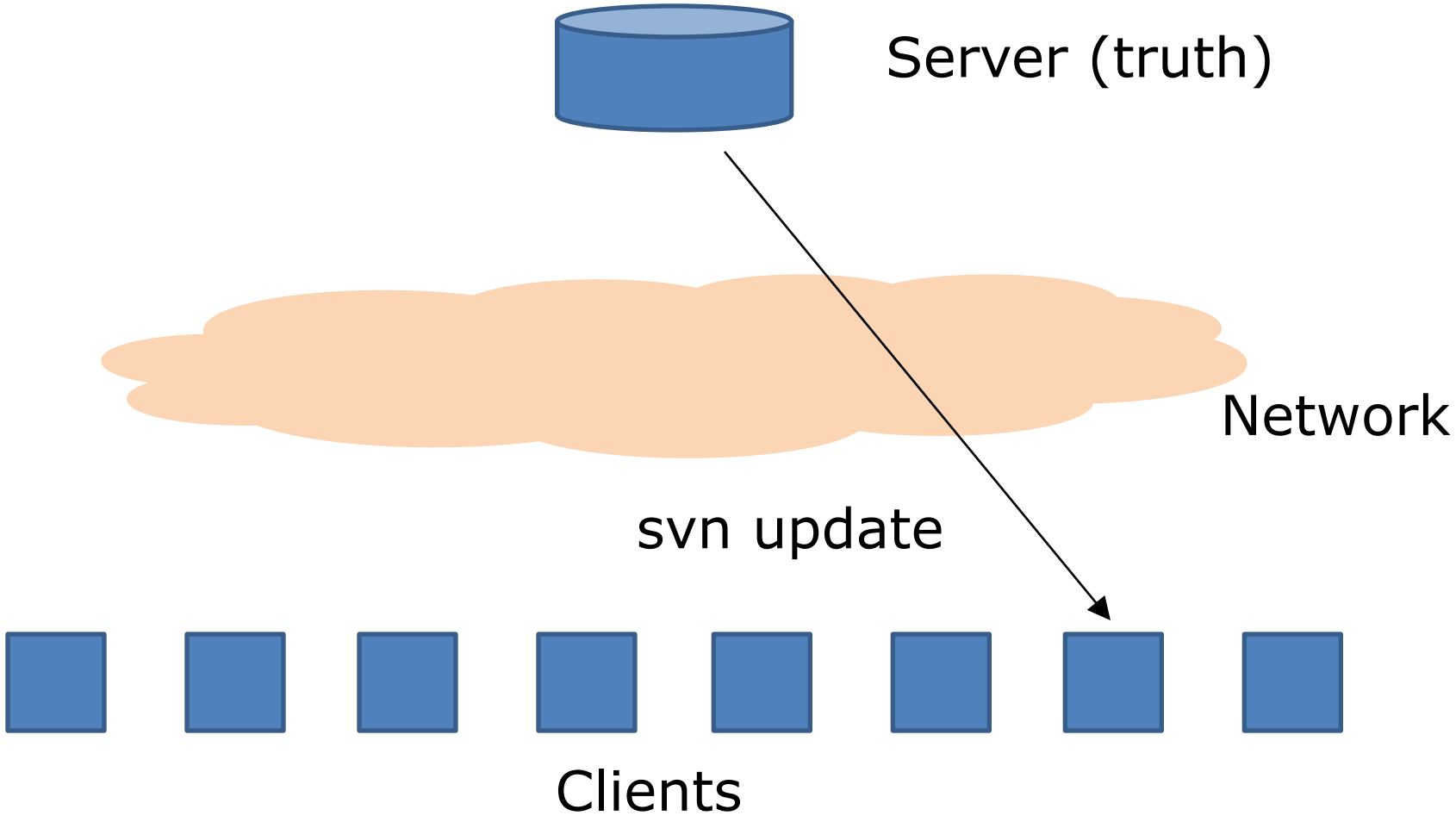
SVN



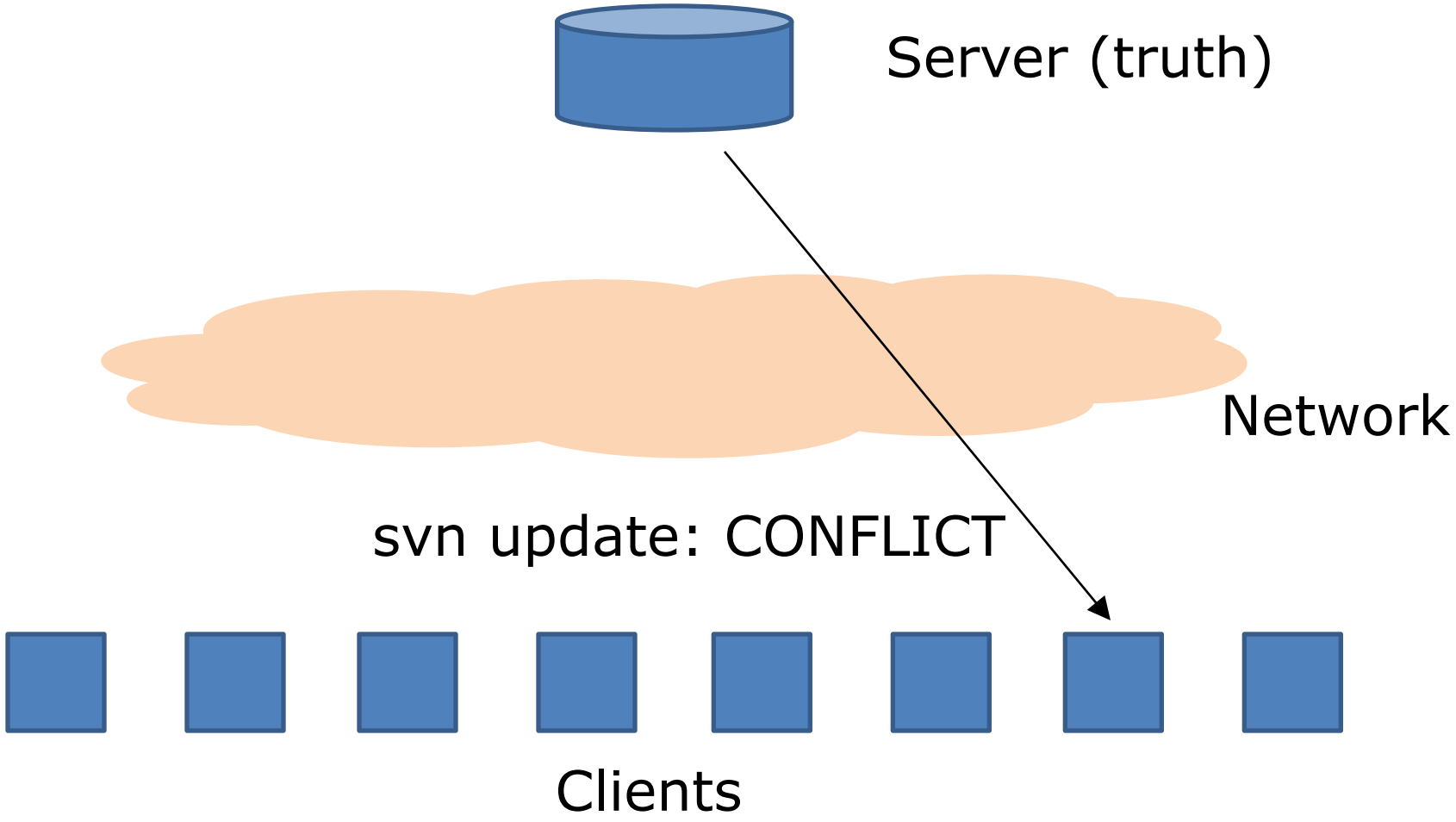
SVN



SVN



SVN



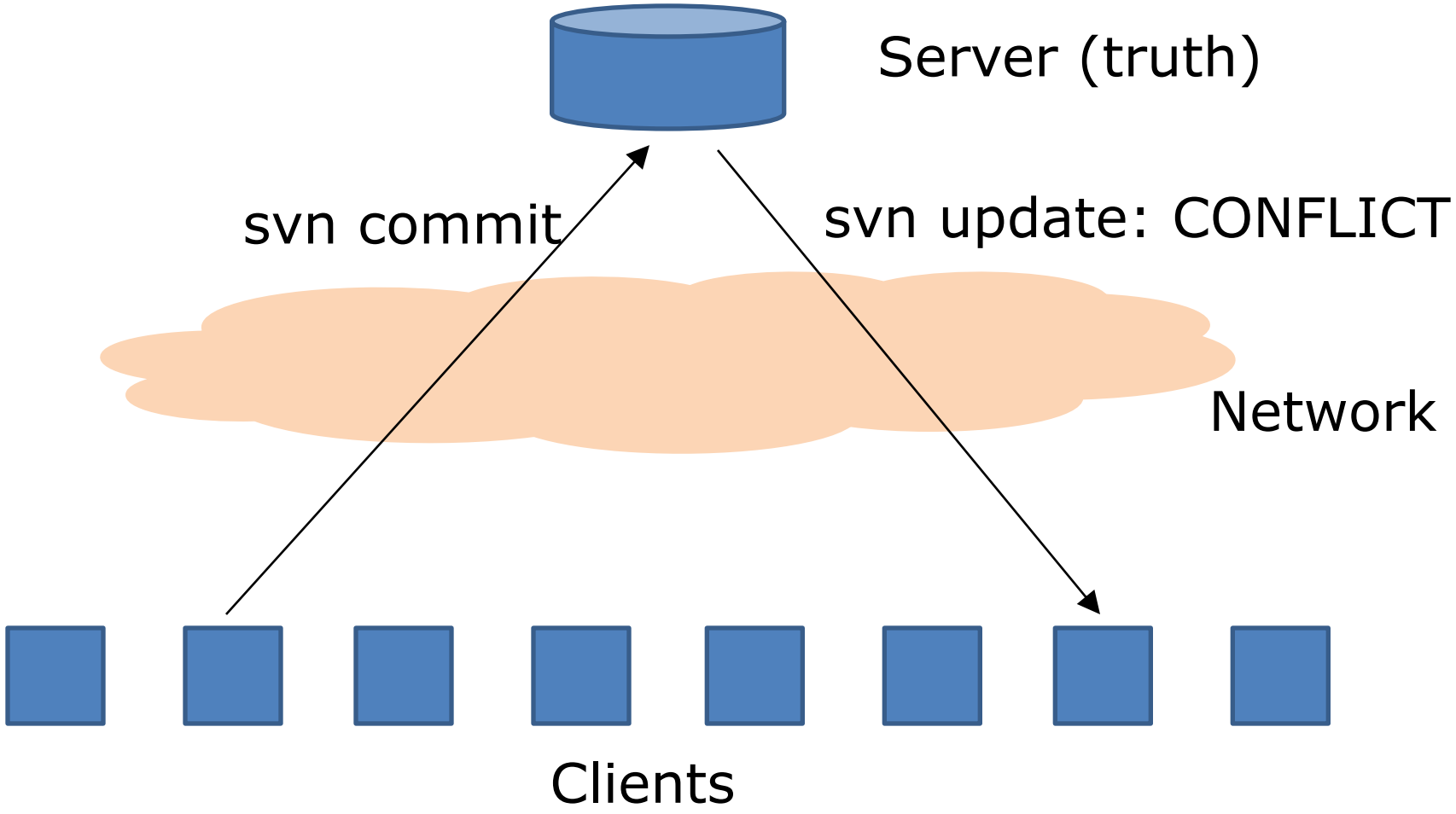
Centralized version control

- Advantages:
  - Everyone knows what everyone else is doing (mostly)
  - Administrators have more fine-grained control
- Disadvantages:
  - Single point of failure
  - Cannot work offline
  - Slow
  - Does not scale

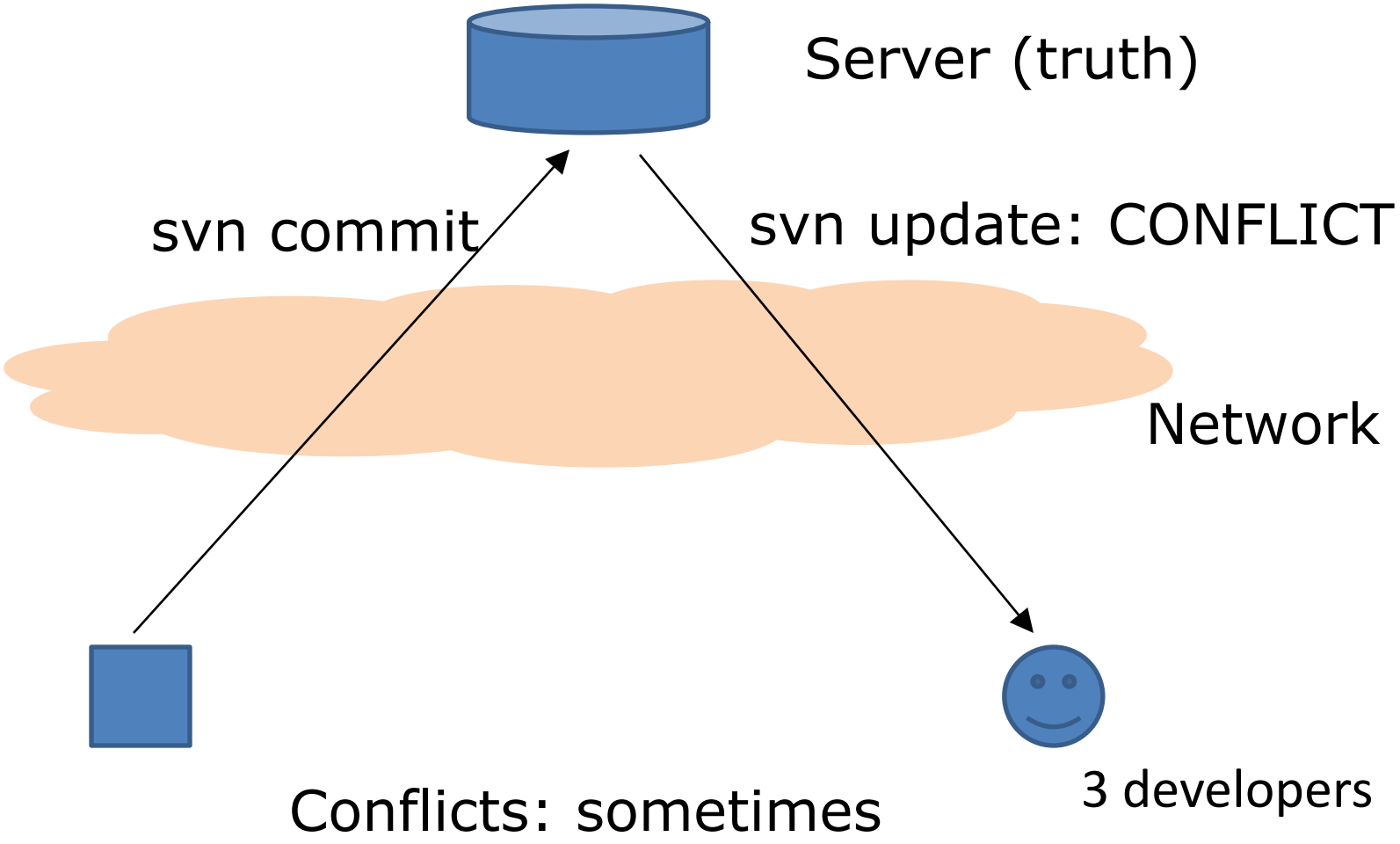
- Easier to lose data
- Incentive to use version control sparingly
- Tangled instead of atomic commits

SVN

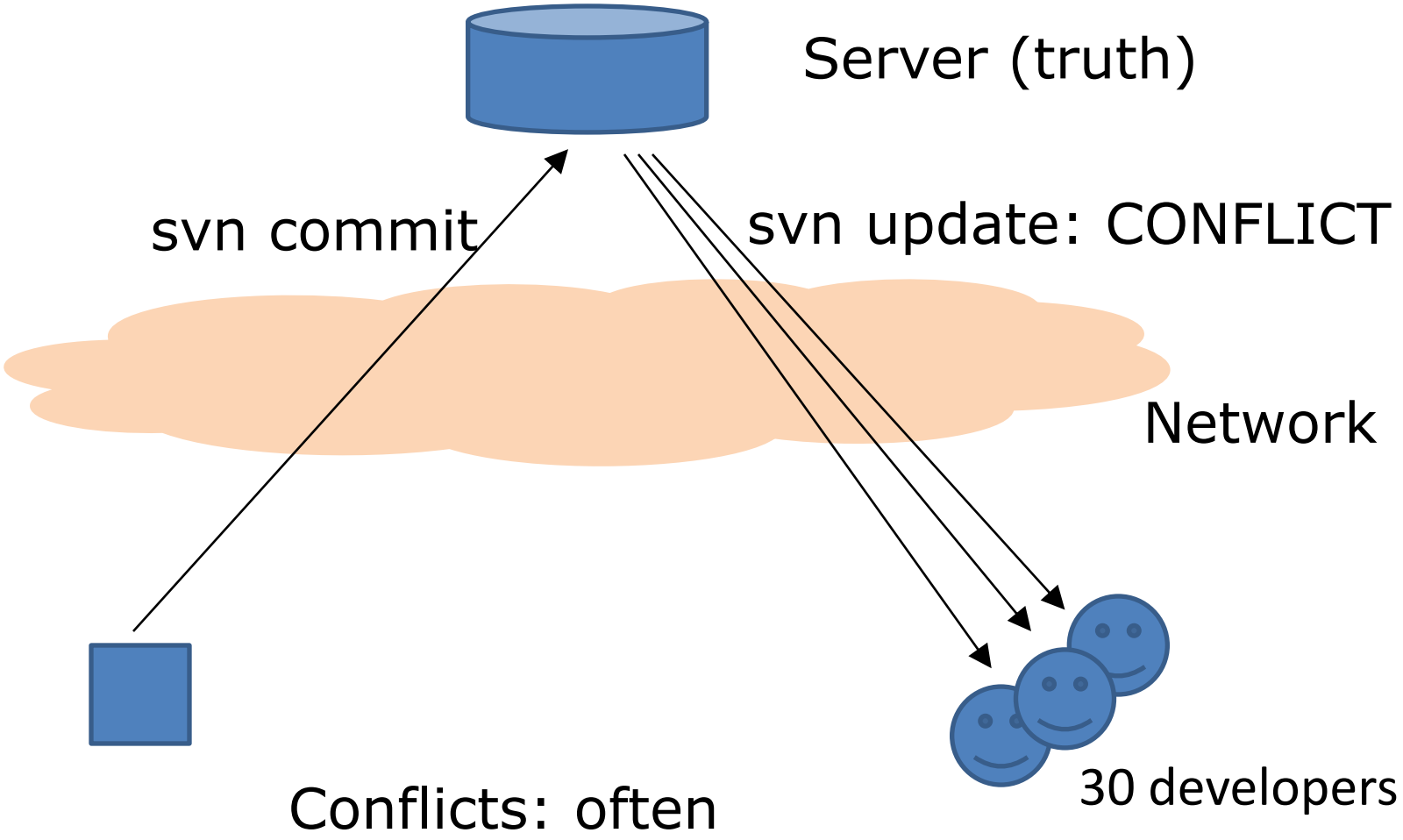
Every time there is a commit on the system there is a chance of creating a conflict with someone else



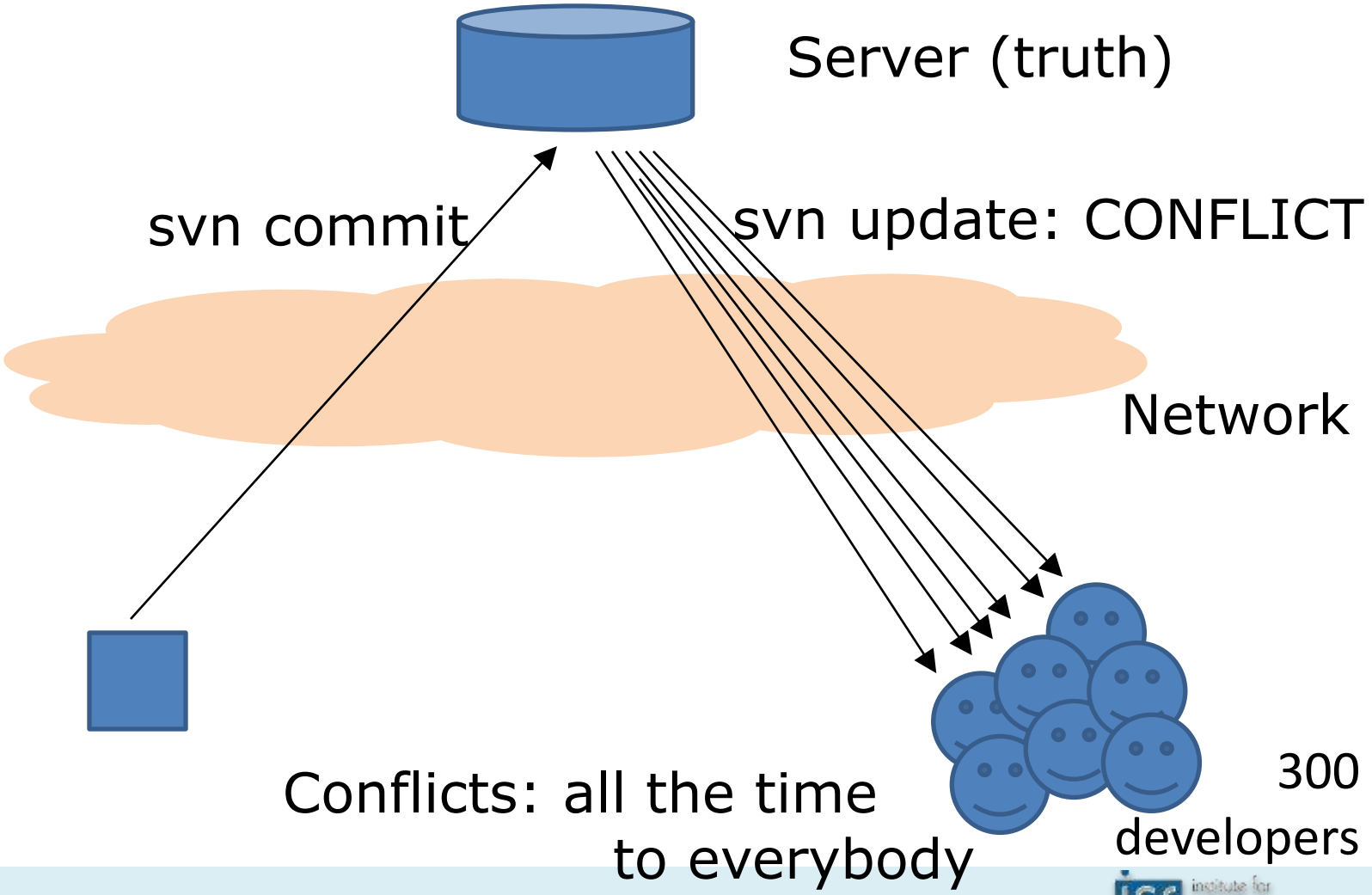
SVN


Server (truth)
svn commit
svn update: CONFLICT
Network
3 developers
Conflicts: sometimes

SVN


Server (truth)
svn commit
svn update: CONFLICT
Network
30 developers
Conflicts: often

SVN


Server (truth)
svn commit
svn update: CONFLICT
Network
Conflicts: all the time
to everybody
300
developers

Brief timeline of VCS

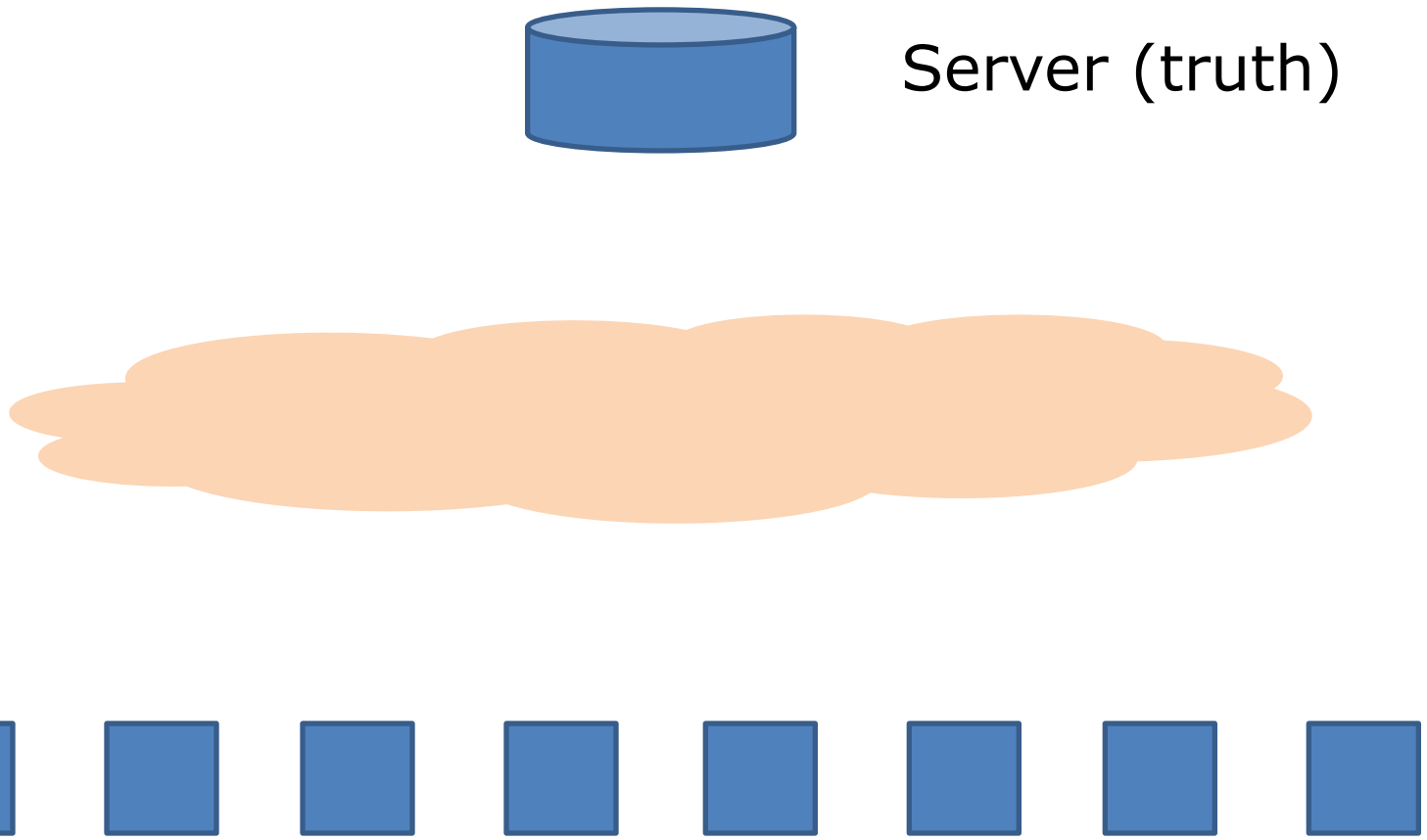
- 1982: RCS (Revision Control System), still maintained
- 1990: CVS (Concurrent Versions System)
- 2000: SVN (Subversion)
- 2005: Bazaar, Git, Mercurial

Git

- Developed by Linus Torvalds, the creator of Linux
- Designed to handle large projects like the Linux kernel efficiently
  - Speed
  - Thousands of parallel branches

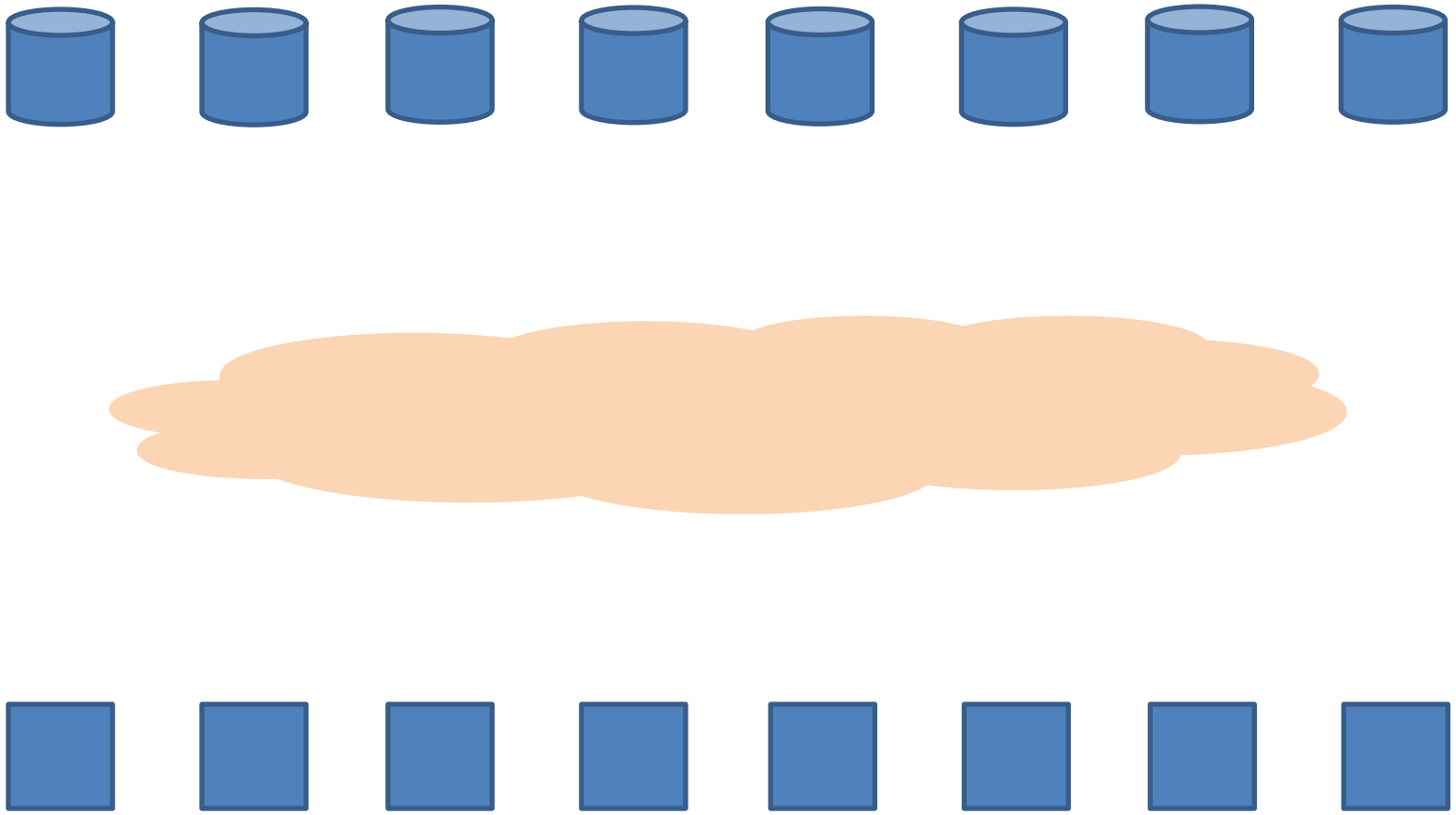
Git

Git is distributed. There is not one server …



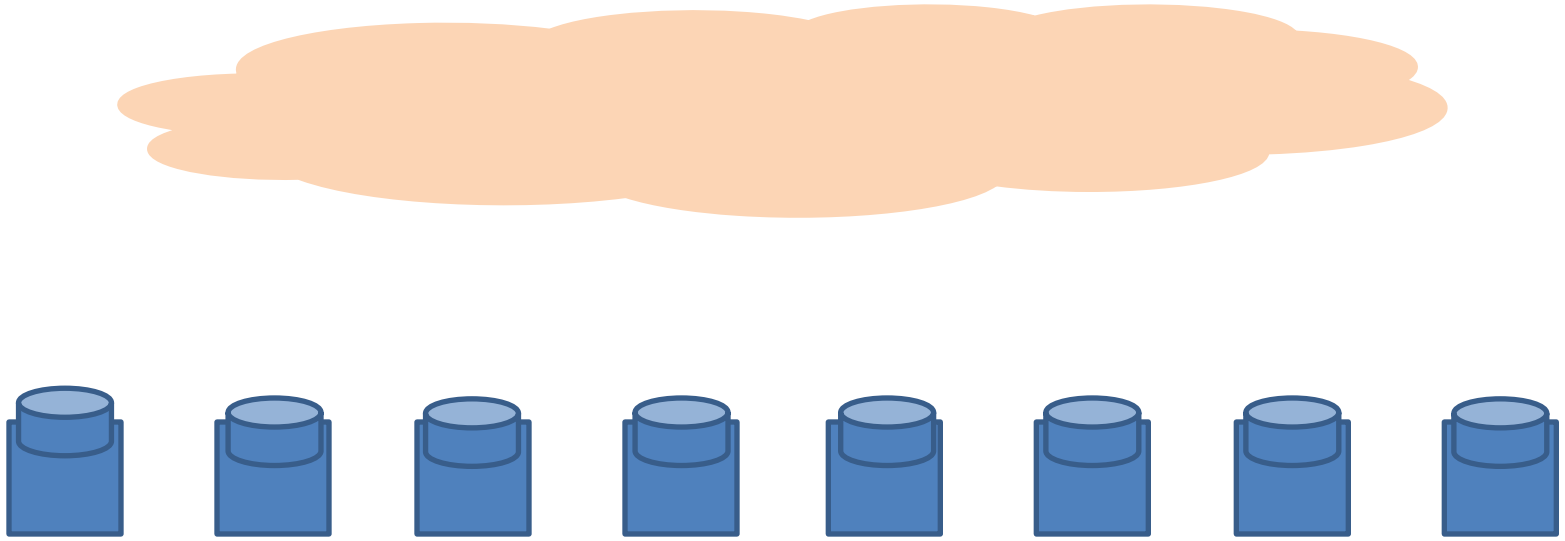
… but many

Git



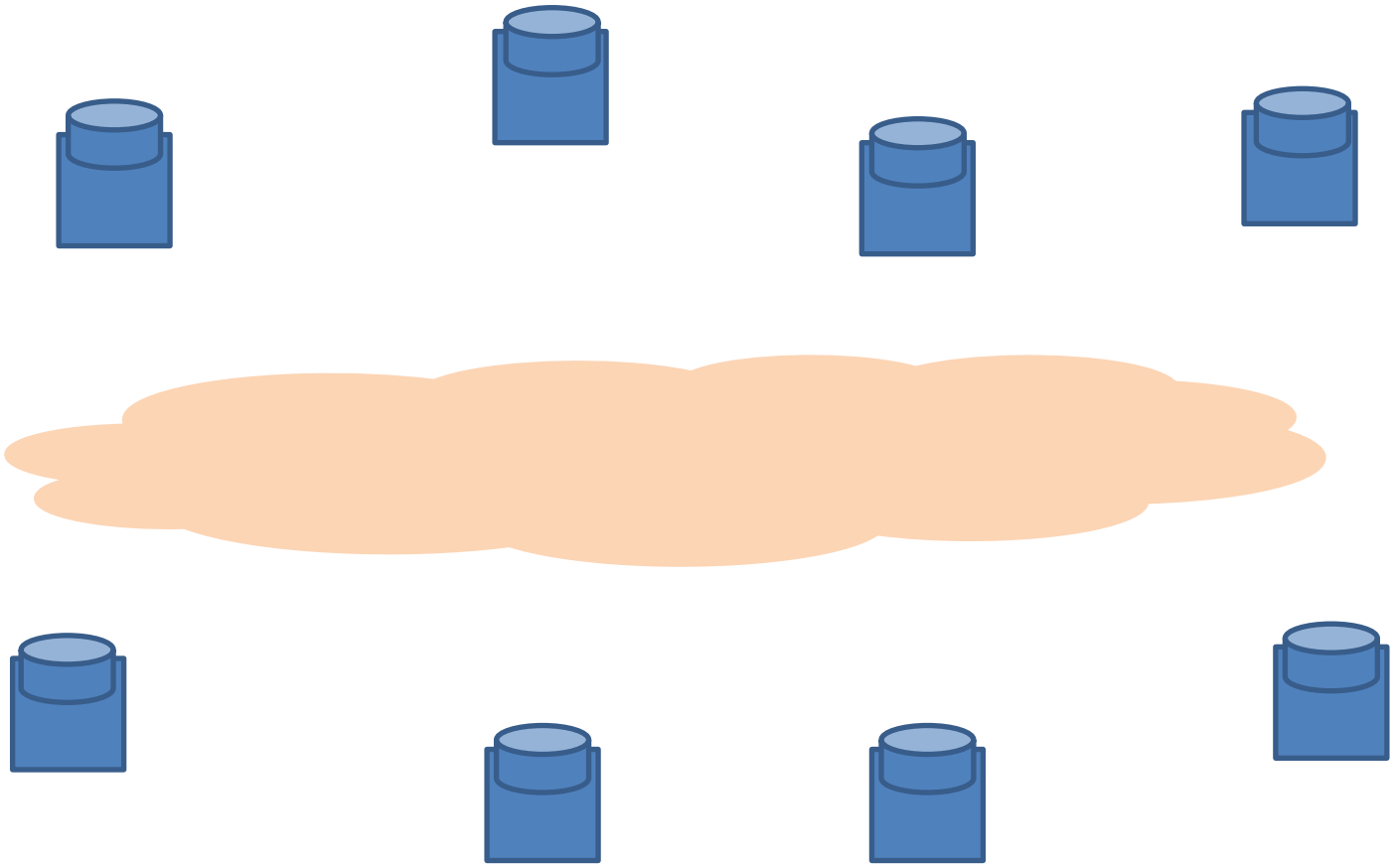
Git

Actually there is one server per computer



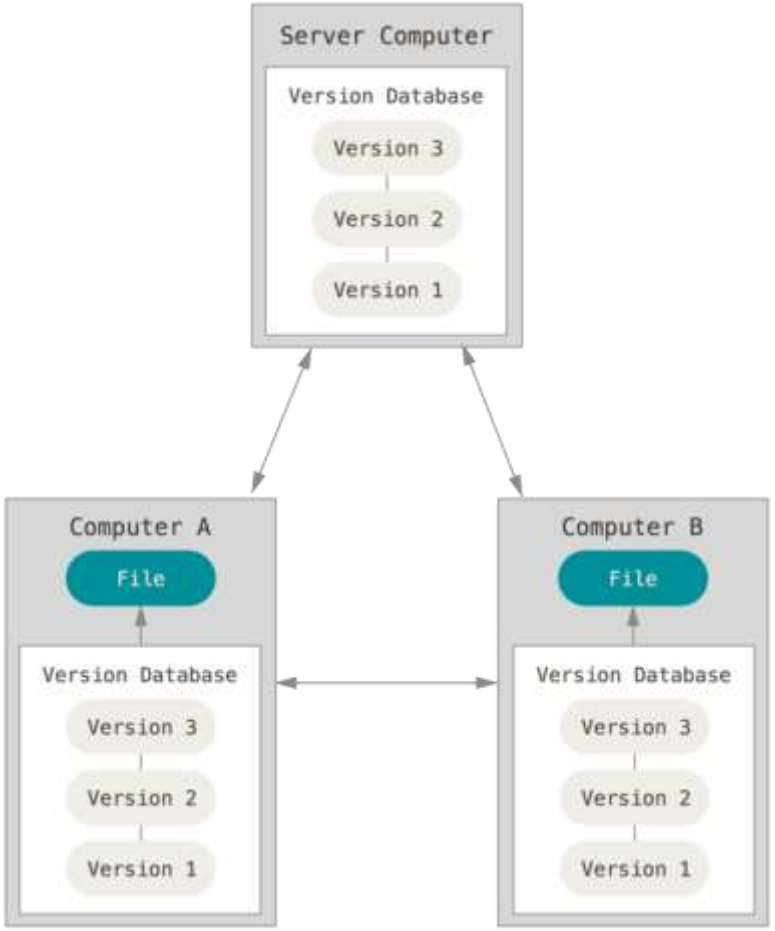
Git

Every computer is a server and version control happens locally.



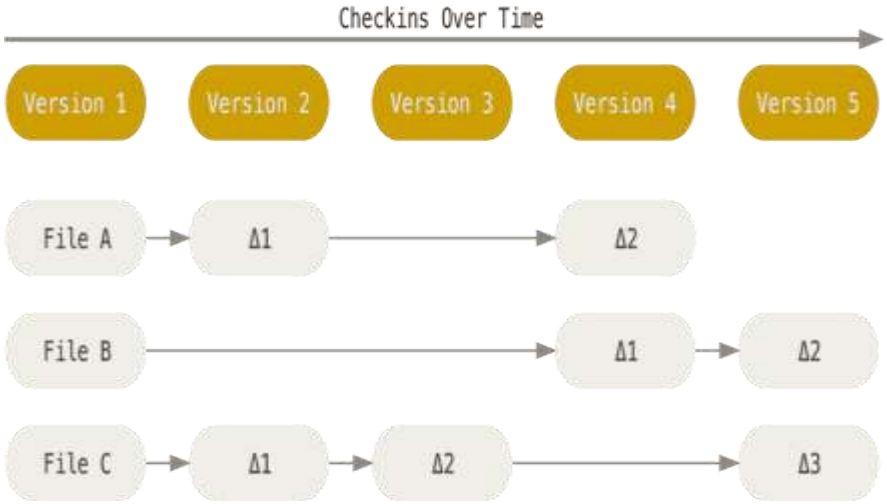
Distributed version control

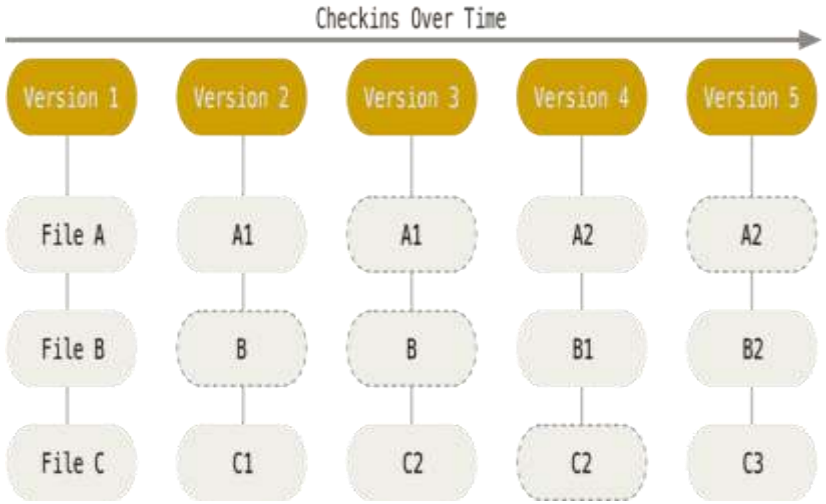
- Clients fully mirror the repository
  - Every clone is a full backup of ***all*** the data
- Advantages:
  - Fast, works offline, scales
  - Better suited for collaborative workflows
- E.g., Git, Mercurial, Bazaar



https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

SVN (left) vs. Git (right)





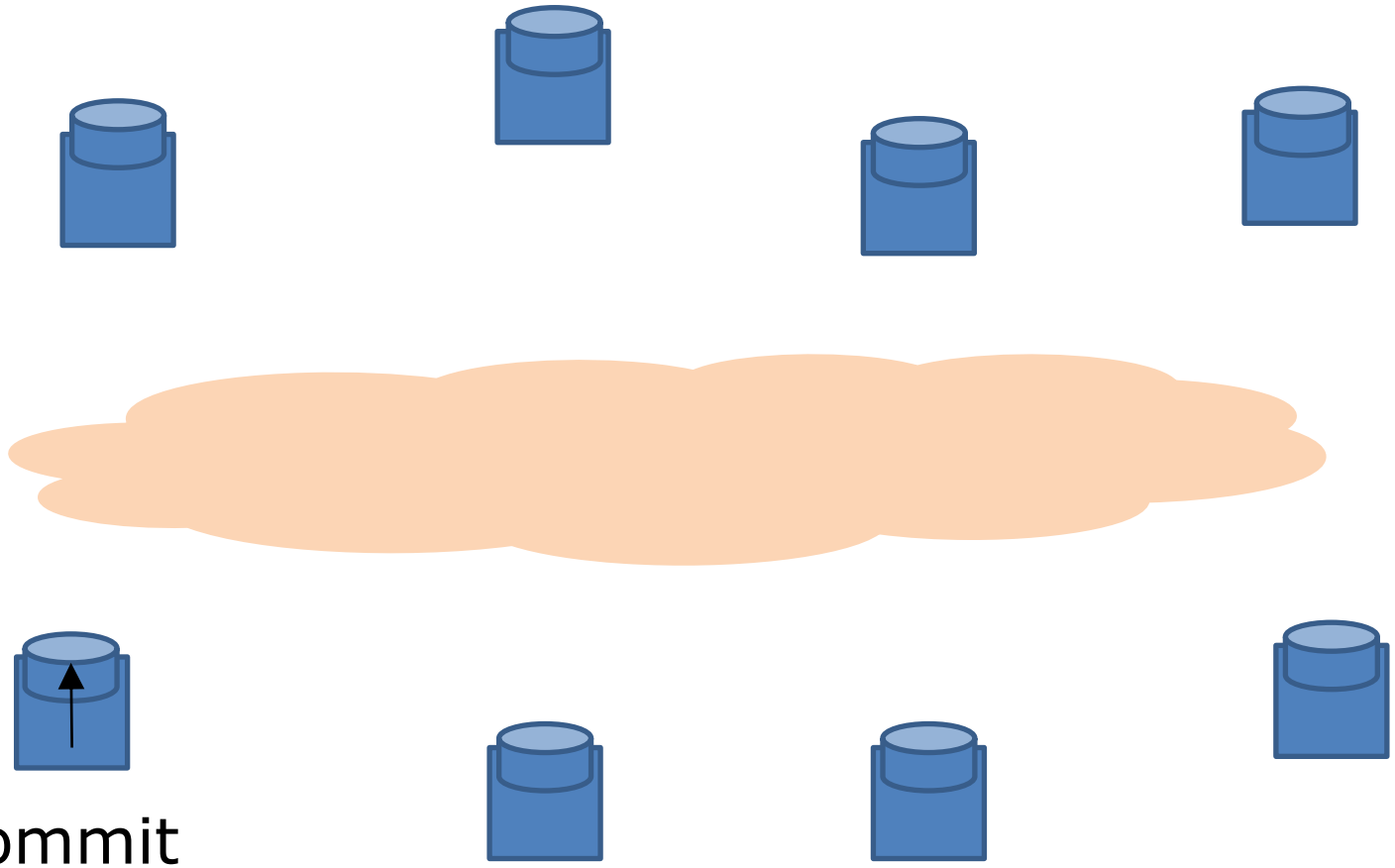
- SVN stores changes to a base version of each file
- Version numbers (1, 2, 3, …) are increased by one after each commit

- Git stores each version as a snapshot
- If files have not changed, only a link to the previous file is stored
- Each version is referred by the SHA-1 hash of the contents

https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Git

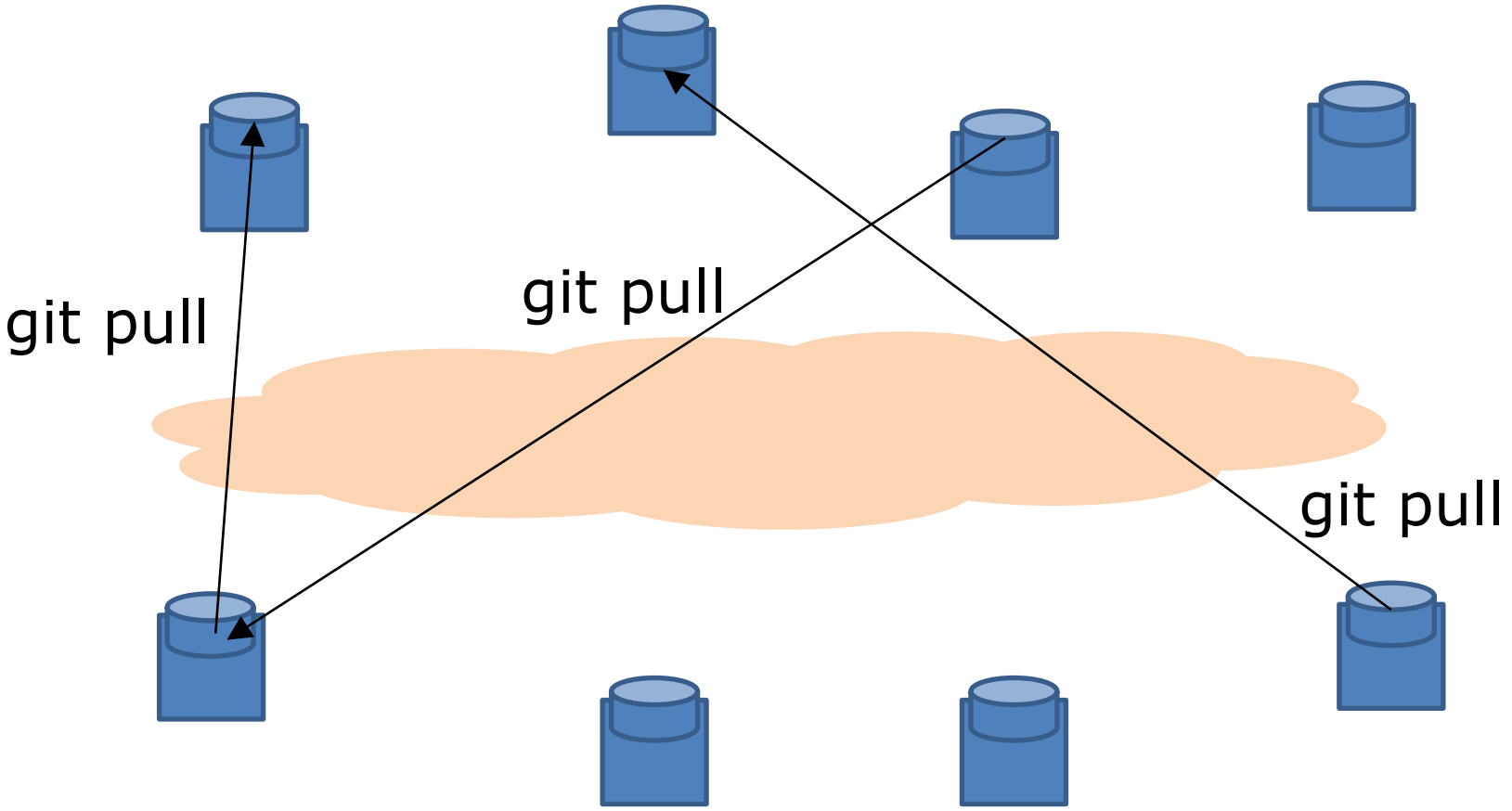
How do you share code with collaborators if commits are *local*?



git commit

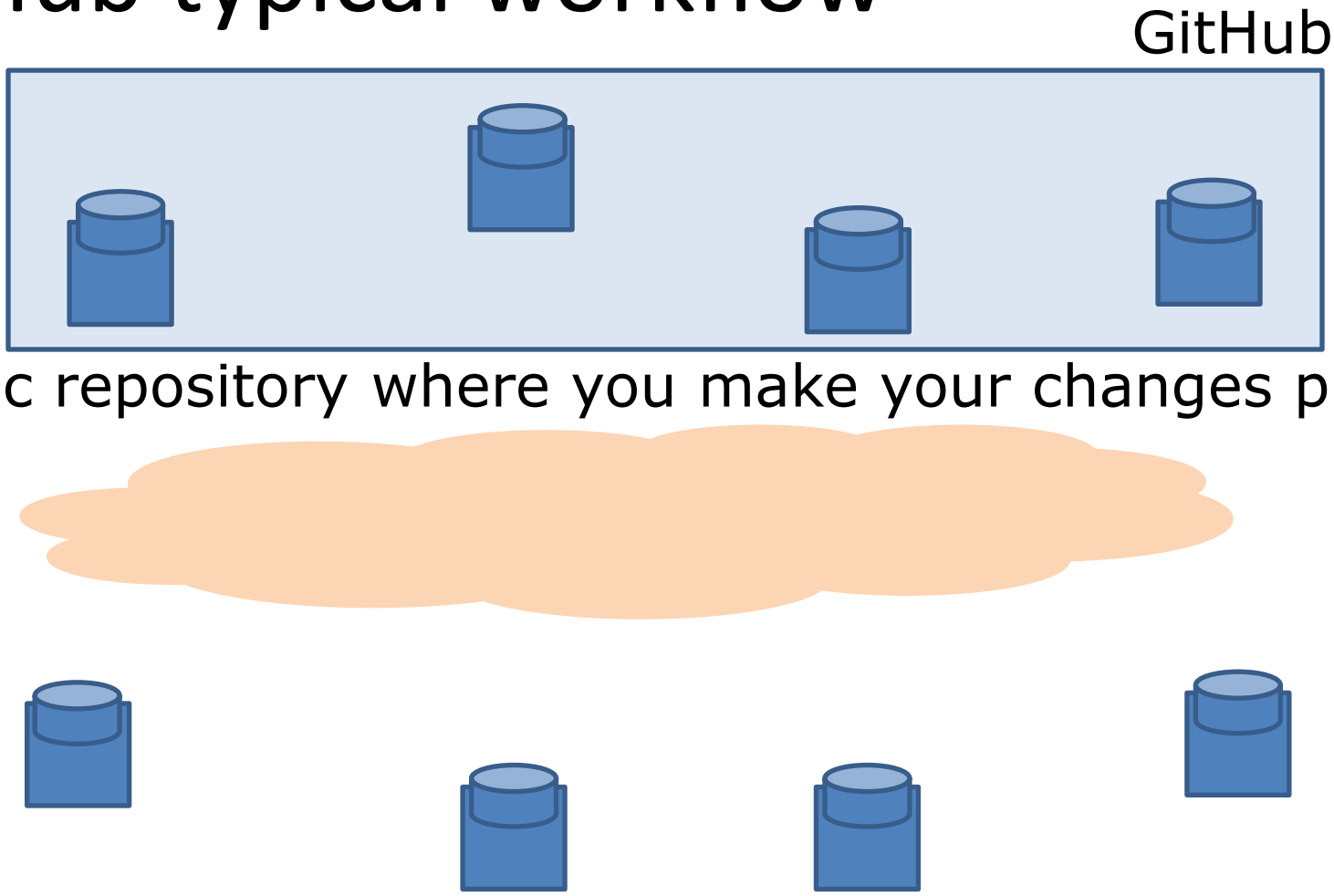
Git

You *pull* their commits from them (and they do the same with yours)

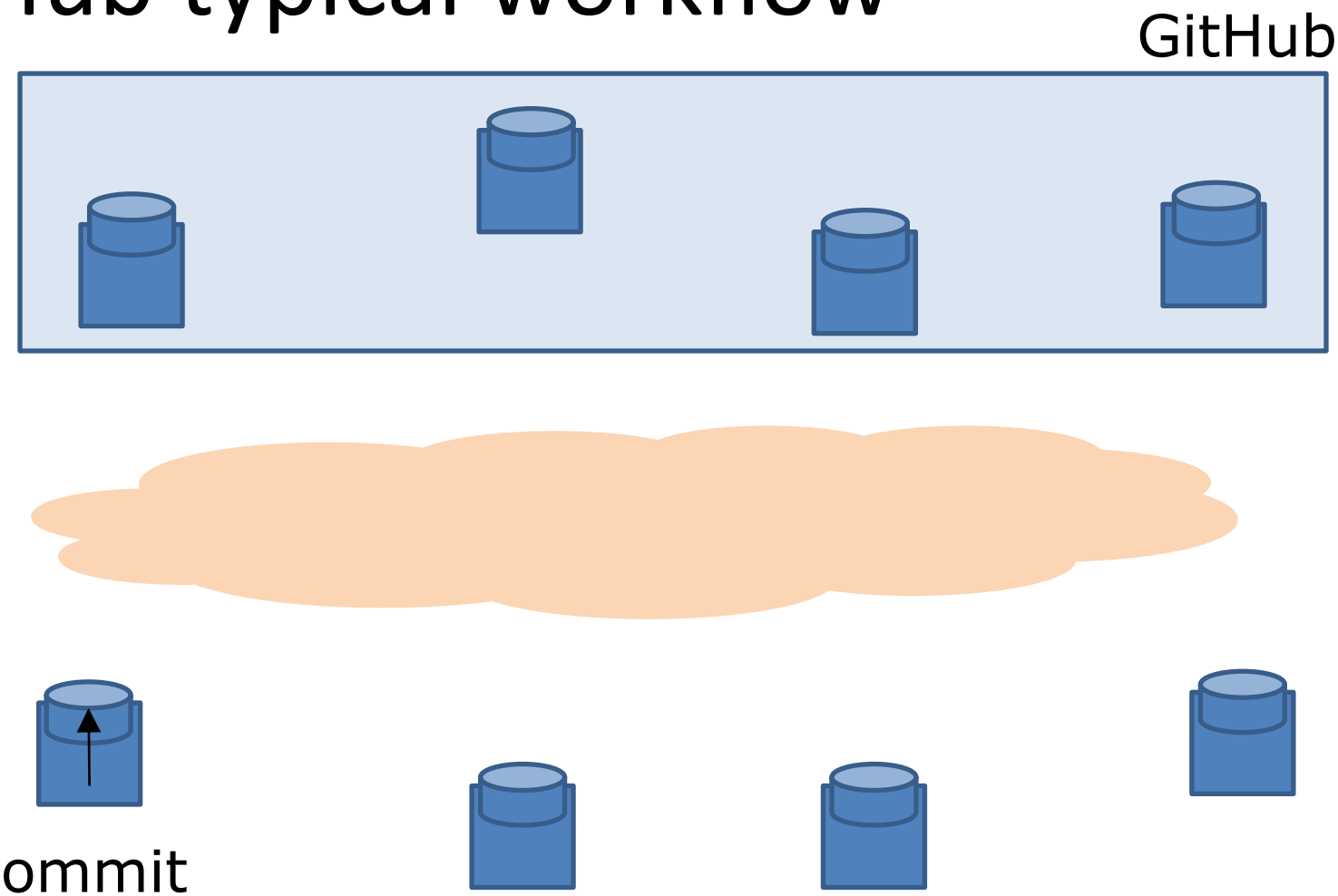


… But requires host names / IP addresses

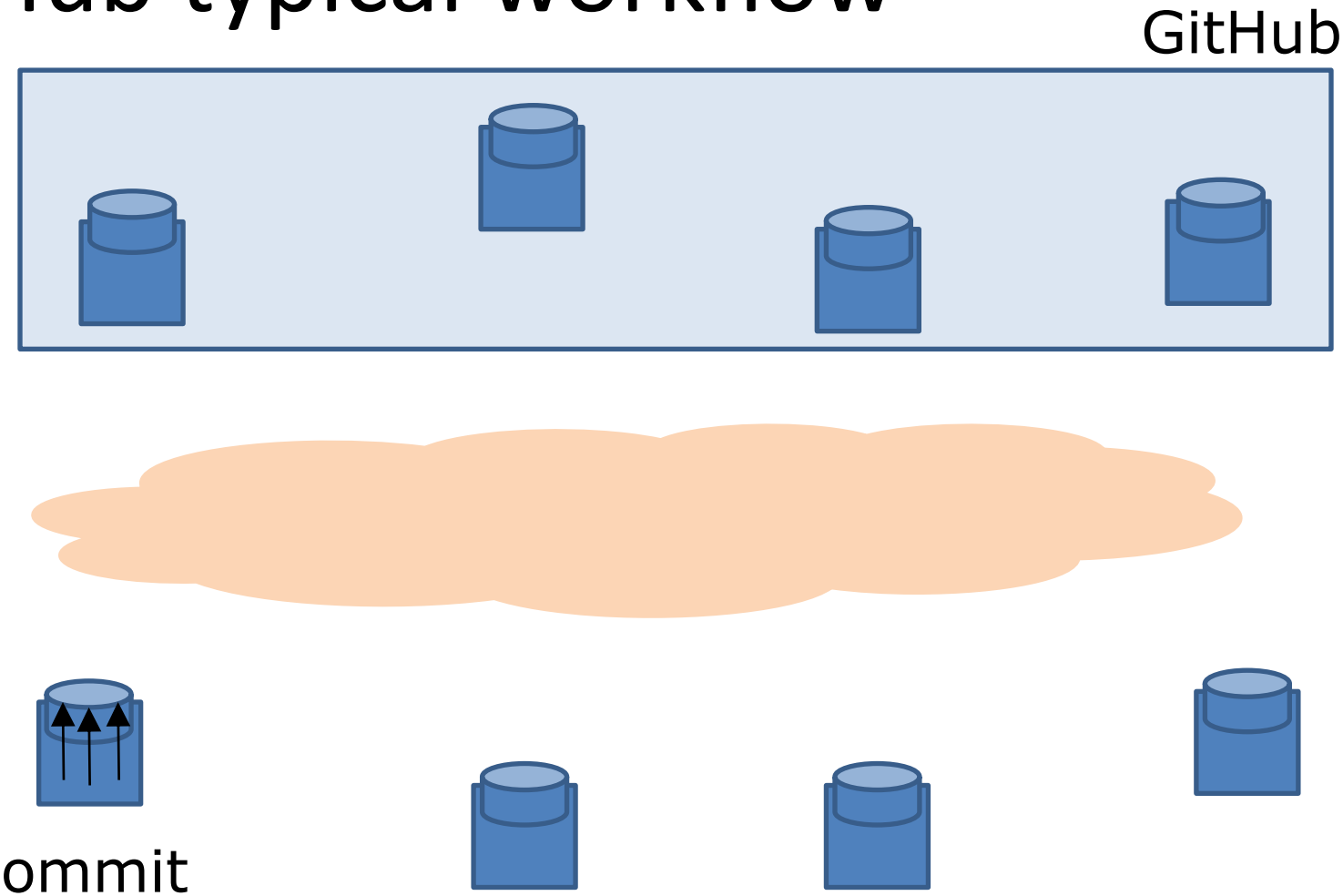
GitHub typical workflow



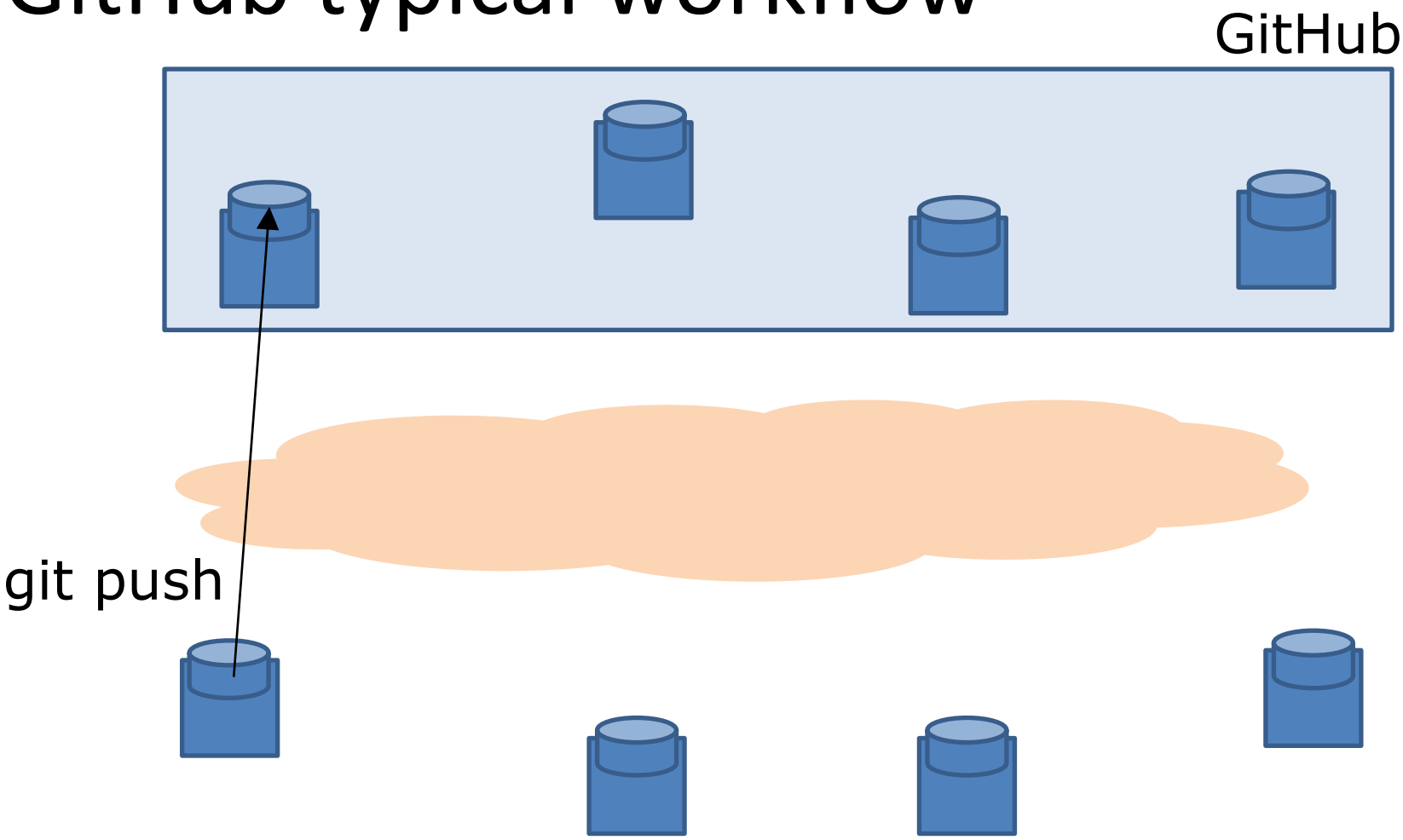
GitHub typical workflow



GitHub typical workflow

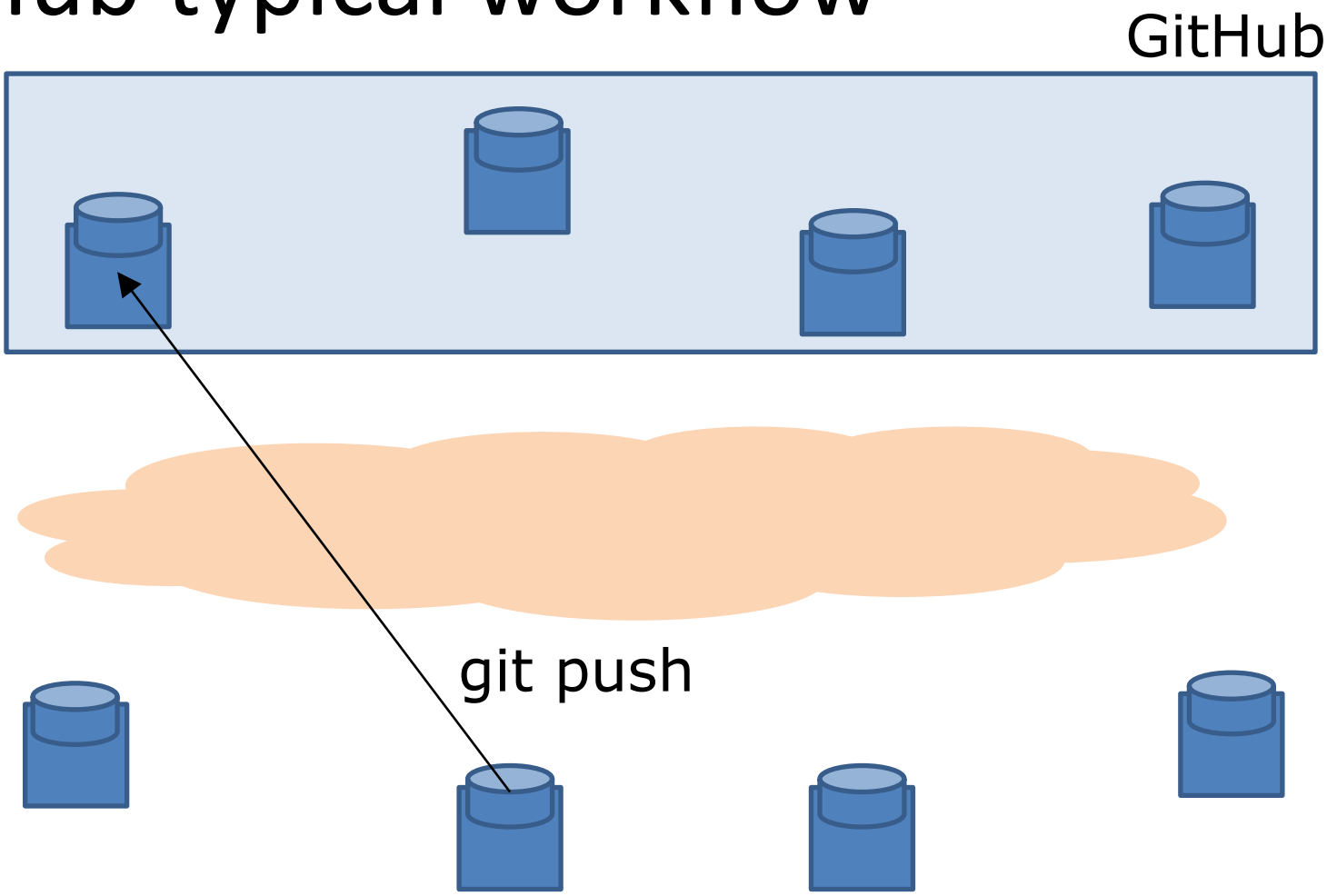


GitHub typical workflow



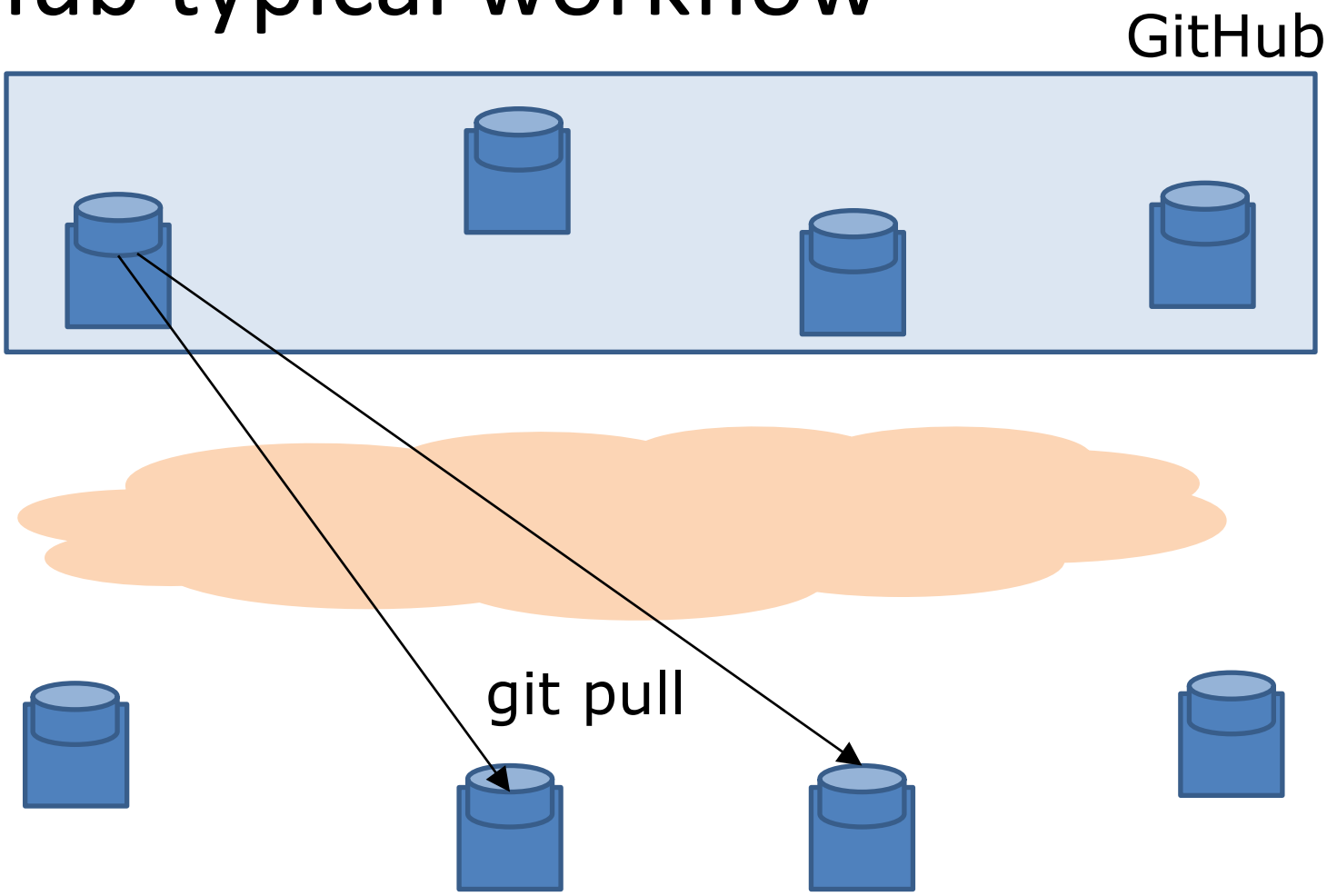
push your local changes into a remote repository.

GitHub typical workflow



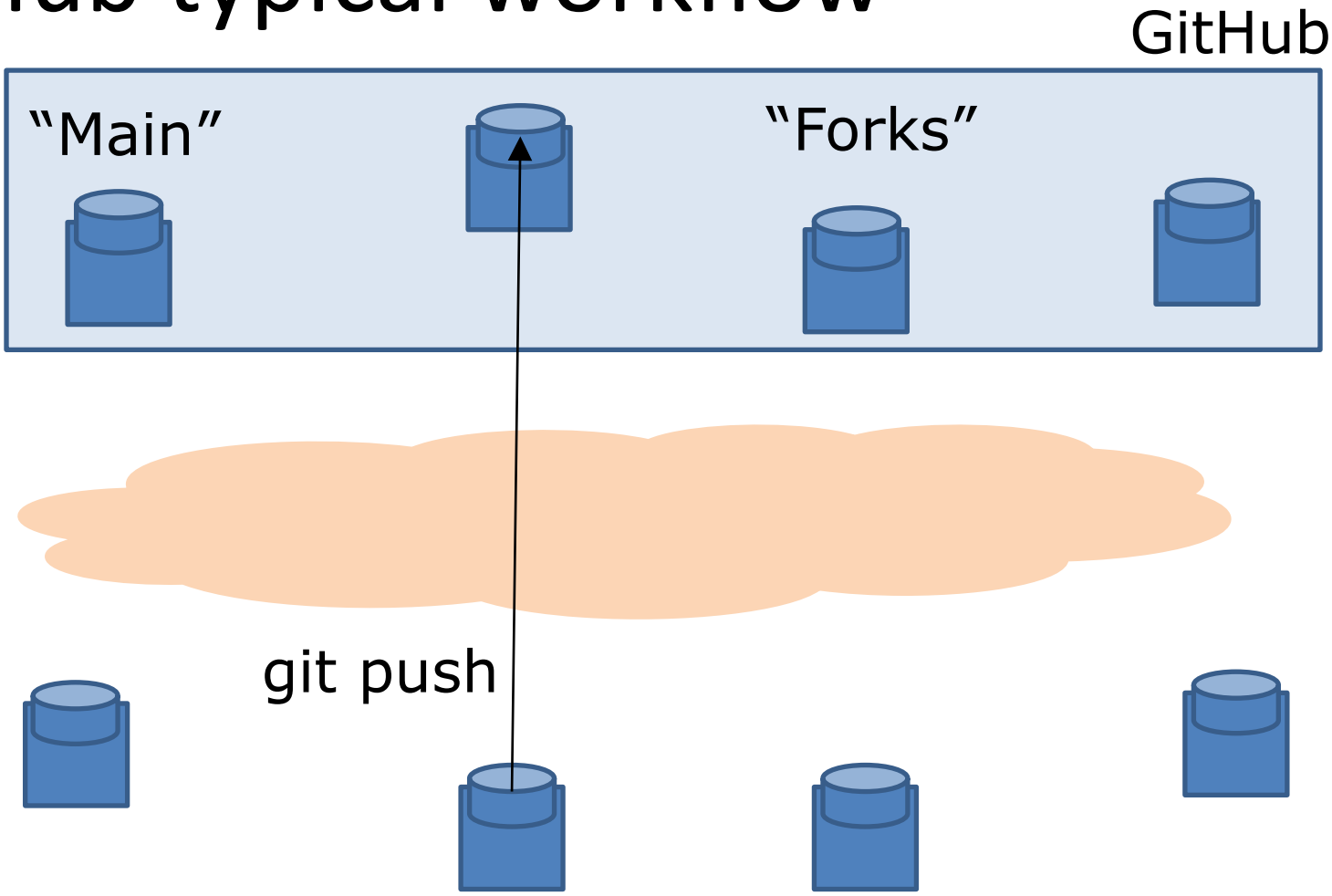
Collaborators can push too if they have access rights.

GitHub typical workflow



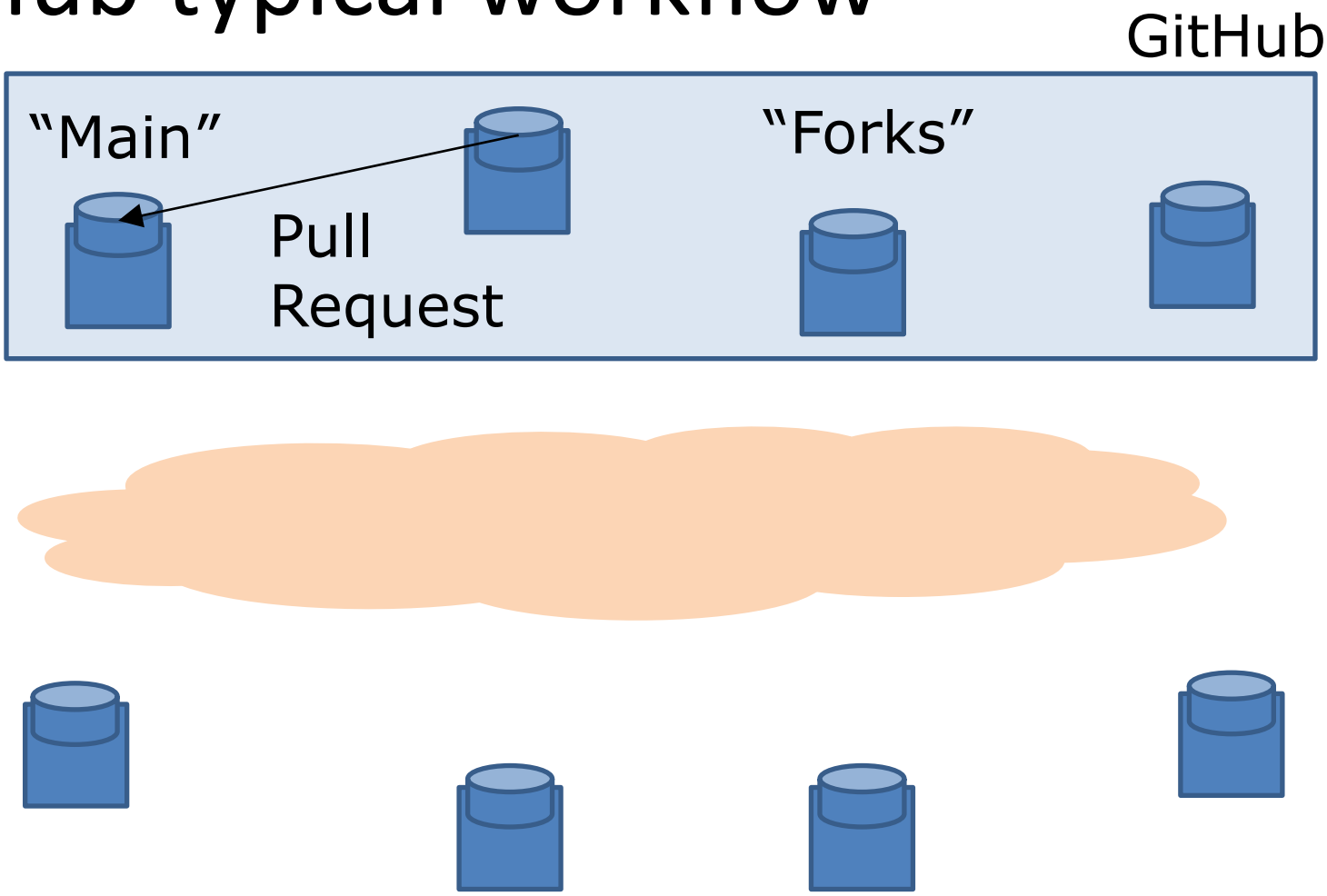
Without access rights, “don’t call us, we’ll call you” (*pull* from trusted sources) … But again requires host names / IP addresses.

GitHub typical workflow



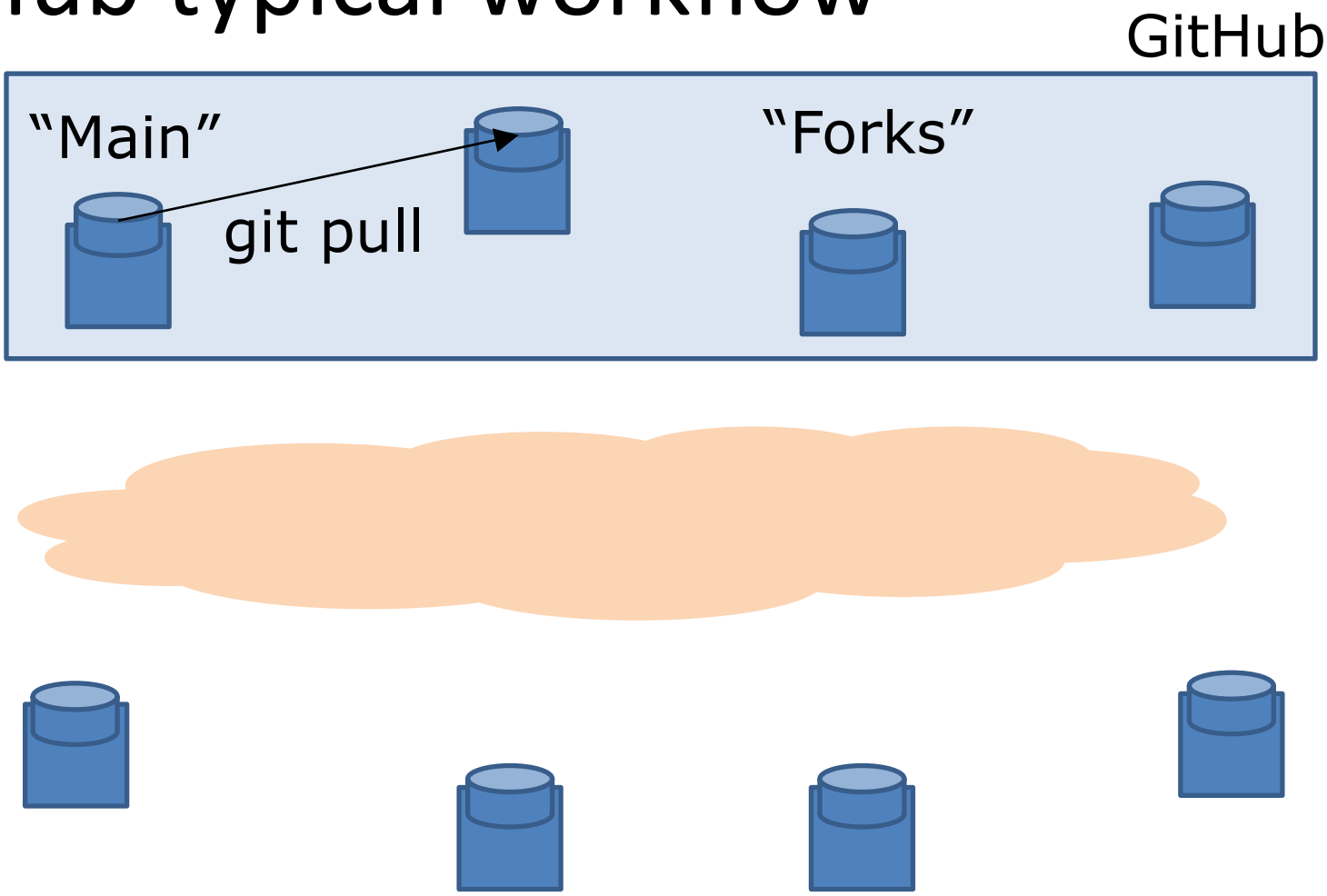
Instead, people maintain public remote "forks" of "main" repository on GitHub and push local changes.

GitHub typical workflow



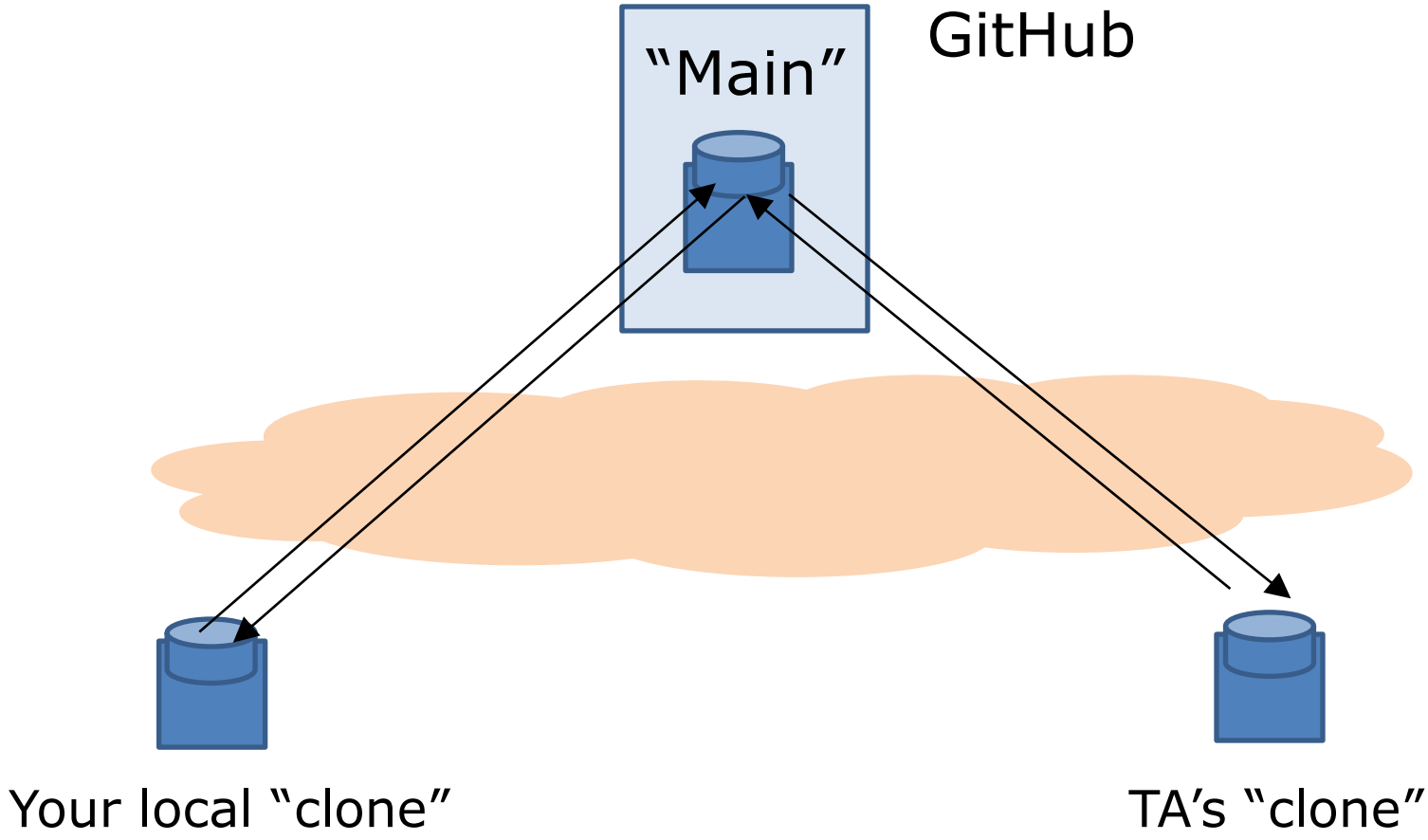
Availability of new changes is signaled via ”Pull Request”.

GitHub typical workflow



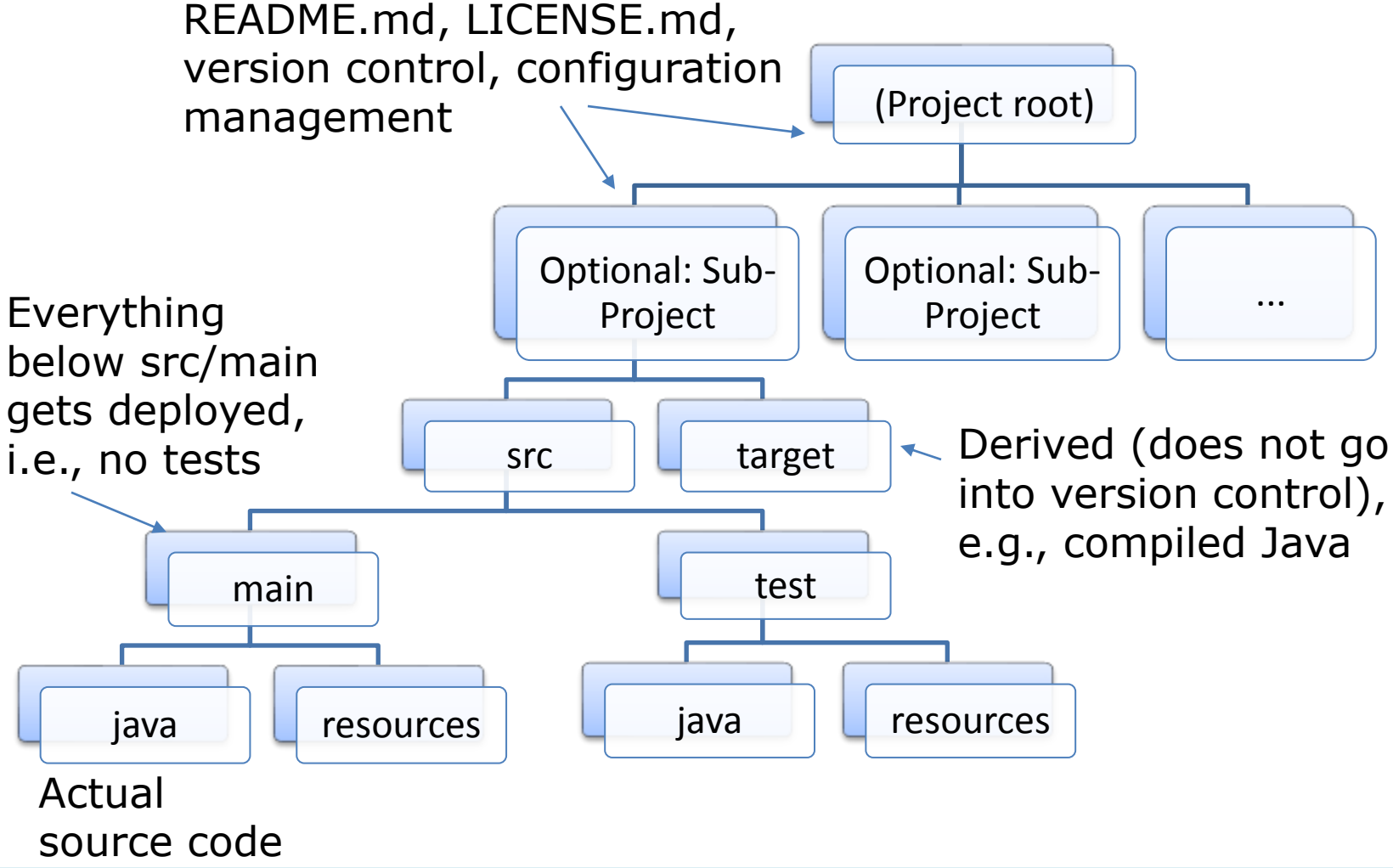
Changes are pulled into main if PR accepted.

214 workflow



You *push* homework solutions; *pull* recitations, homework assignments, grades. TAs vice versa

Organizing a Development Project



Build Manager

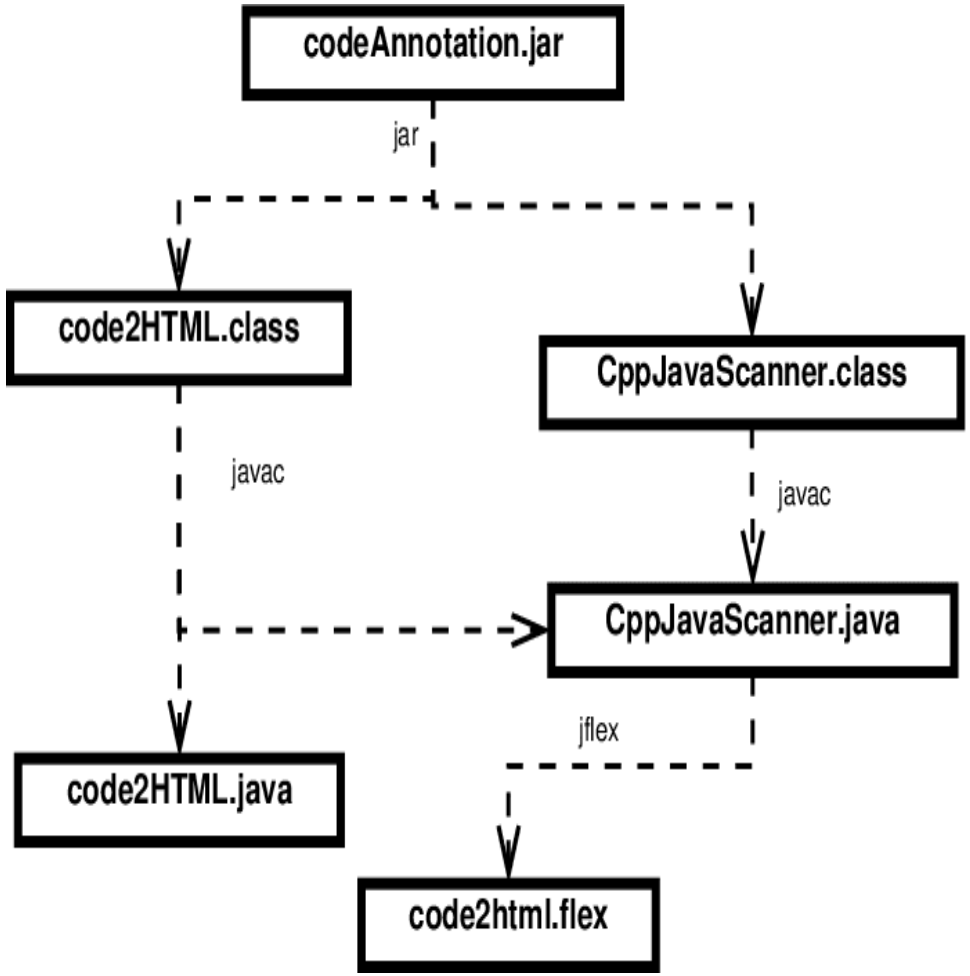
- Tool for scripting the automated steps required to produce a software artifact, e.g.:
  - Compile Java files in src/main/java, place results in target/classes
  - Compile Java files in src/test/java, place results in target/test-classes
  - Run JUnit tests in target/test-classes
  - If all tests pass, package compiled classes in target/classes into .jar file.

Types of Build Managers

- IDE project managers (limited functionality)
- Dependency-Based Managers
  - Make (1977)
- Task-Based Managers
  - Ant (2000)
  - Maven (2002)
  - Ivy (2004)
  - **Gradle** (2012)

Dependency-Based Managers

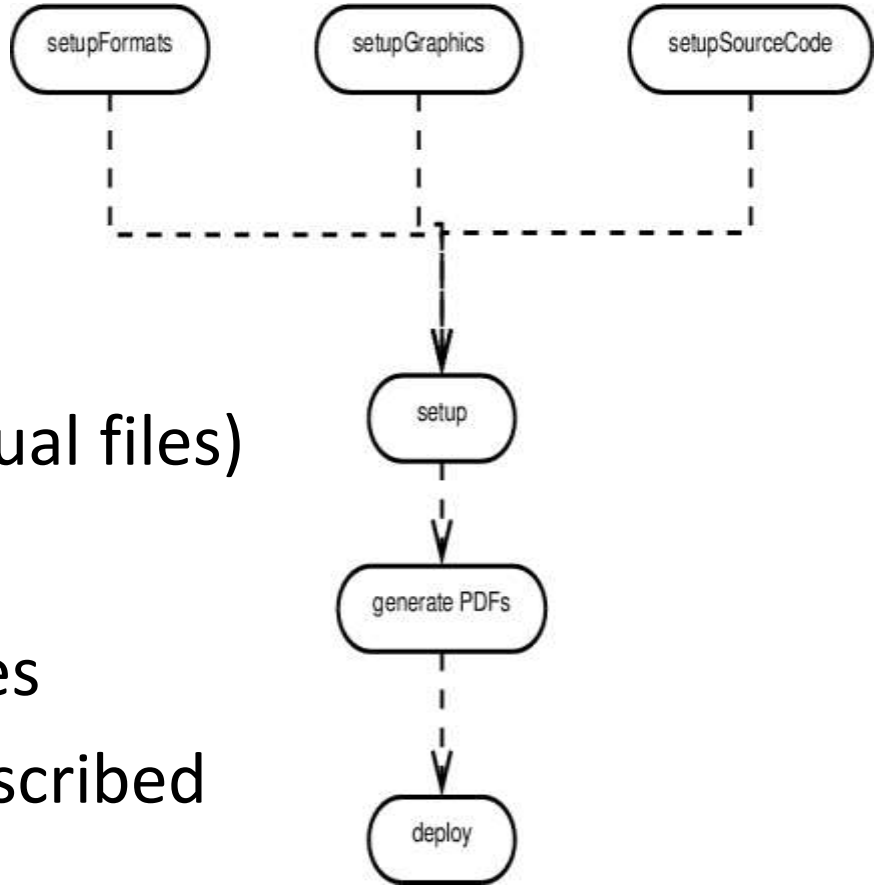
- Dependency graph:
  - Boxes: files
  - Arrows: dependencies; "A depends on B": if B is changed, A must be regenerated
- Build manager (e.g., Make) determines min number of steps required to rebuild after a change.



From: https://www.cs.odu.edu/~zeil/cs350/s17/Public/buildManagers/index.html

Task-Based Managers: Ant

- Disadvantages of Make:
  - Not portable (system-dependent commands, paths, path lists)
  - Low level (focus on individual files)
- Ant:
  - Focus on task dependencies
  - Targets (dependencies) described in build.xml



From: https://www.cs.odu.edu/~zeil/cs350/s17/Public/buildManagers/index.html

Task-Based Managers: Maven

- Maven:
  - build management (like Ant),
  - and configuration management (unlike Ant)
- Can express standard project layouts and build conventions (project archetypes)
- Still uses XML (pom.xml)

Task-Based Managers: Gradle

- Combines the best of Ant and Maven
- From Ant keep:
  - Portability: Build commands described platform-independently
  - Flexibility: Describe almost any sequence of processing steps
- … but drop:
  - XML as build language, inability to express simple control flow
- From Maven keep:
  - Dependency management
  - Standard directory layouts & build conventions for common project types
- … but drop:
  - XML, inflexibility, inability to express simple control flow

Big Builds

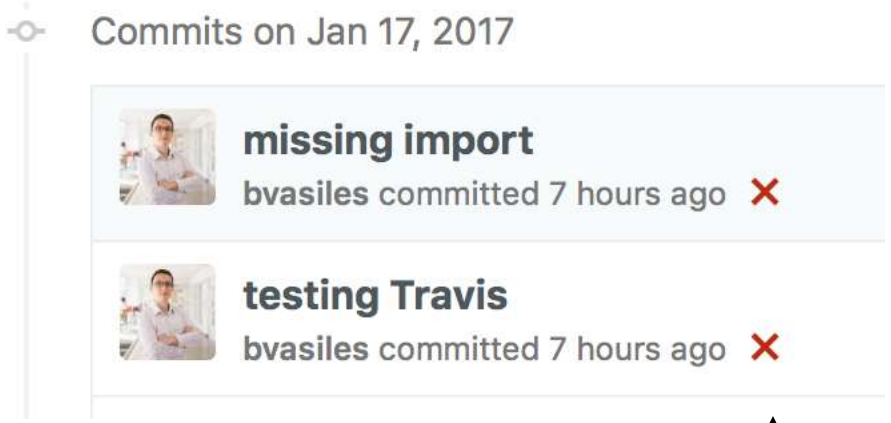
- Must run frequently:
  - fetching and setup of 3rd party libraries
  - static analysis
  - compilation
  - unit testing
  - packaging of artifacts
- Can run less frequently:
  - documentation
  - deployment
  - integration testing
  - test coverage reporting
  - system testing
- Keep track of different Ant/Maven targets, or …

Continuous Integration

- Version control with central "official" repository. Run:
  - automated builds & tests (unit, integration, system, regression) **with every change** (commit / pull request)
  - Test, ideally, in clone of *production* environment
  - E.g., Jenkins (local), Travis CI (cloud-based)

- Advantages:
  - Immediate testing of all changes
  - Integration problems caught early and fixed fast
  - Frequent commits encourage modularity
  - Visible code quality metrics motivate developers
  - (cloud-based) Local computer not busy while waiting for build
- Disadvantages:
  - Initial effort to set up

Travis CI



- Cloud-based CI service; GitHub integration
  - Listens to *push* events and *pull request* events and starts “build” automatically
  - Runs in virtual machine / Docker container
  - Notifies submitter of outcome; sets GitHub flag

- Setup: project top-level folder .travis.yml
  - Specifies which environments to test in (e.g., jdk versions)

Coding Standards (Checkstyle)

- Code conventions are important:
  - 80% of software lifetime cost goes to **maintenance**.
  - Most software are not maintained by the **original author**.
  - Code conventions improve software **readability**; maintain **consistency** across teams and time.

- Checkstyle, tool to automate coding standards:
  - Runs as part of automated build / continuous integration
  - Checks, e.g.,:
    - Line Length: e.g., Avoid lines longer than 80 characters
    - Wrapping lines: e.g., Break after comma, before operator
    - Comments: Javadoc
    - Naming conventions: e.g., class names, variable names
    - Indentation: e.g., spaces over tabs