Principles of Software Construction:
Objects, Design, and Concurrency

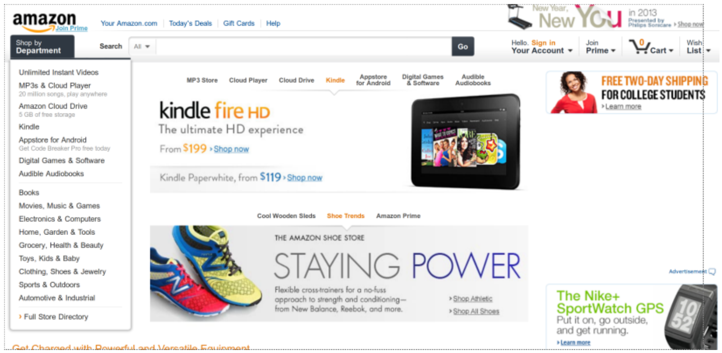
Part 1: Introduction

Course overview and introduction to software design

Charlie Garrod Bogdan Vasilescu

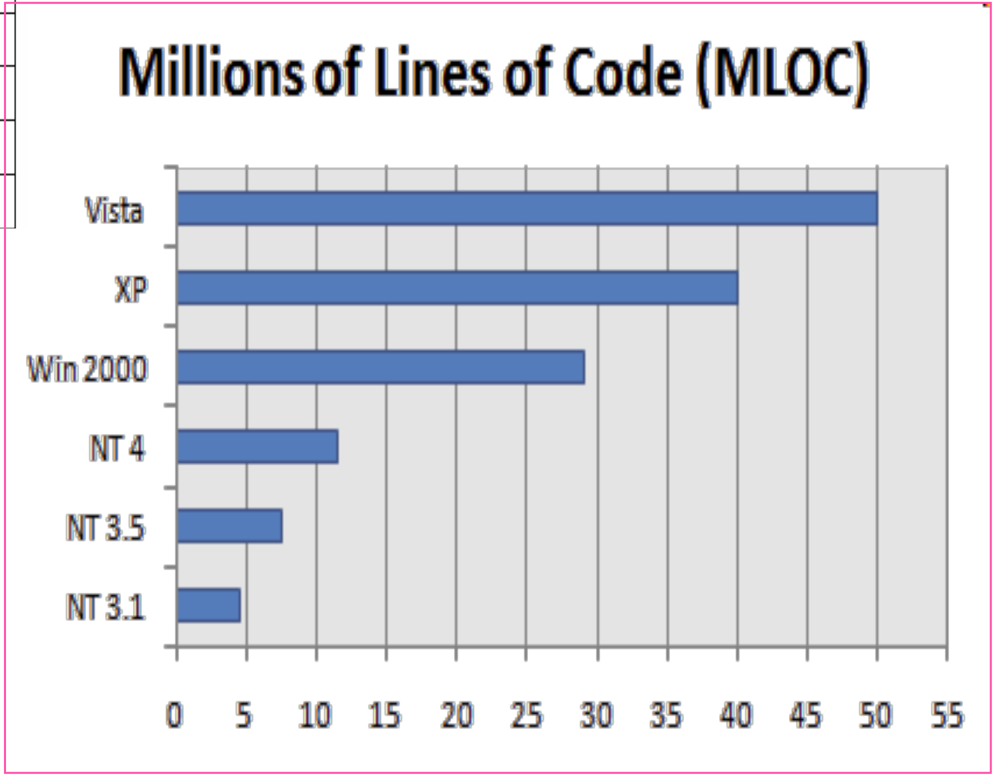
School of Computer Science
institute for SOFTWARE RESEARCH

Software is everywhere



Growth of code and complexity over time

n 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

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.

Normal night-time image

Blackout of 2003

Principles of Software Construction:
Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Charlie Garrod Bogdan Vasilescu

School of
Computer Science

institute for
SOFTWARE
RESEARCH

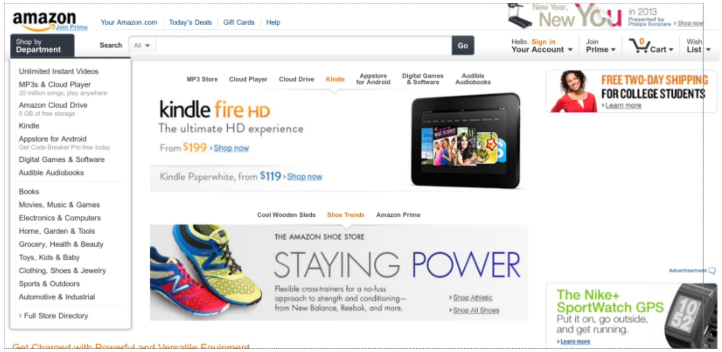
primes

graph search

binary tree

GCD

sorting



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 the **design principles** for construction of software systems

Principles of Software Construction:
Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Charlie Garrod Bogdan Vasilescu

School of
Computer Science

institute for
SOFTWARE
RESEARCH

Objects in the real world

Object-oriented programming

- Programming based on structures that contain both data and methods

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

  public Bicycle(…) { … }

  public void accelerate() {
    speed++;
  }

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

Principles of Software Construction:
Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Charlie Garrod Bogdan Vasilescu

School of Computer Science

institute for SOFTWARE RESEARCH

institute for SOFTWARE RESEARCH

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

- 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 a configurable order, version A

```
static void sort(int[] list, boolean ascending) {
   …
   boolean mustSwap;
   if (ascending) {
      mustSwap = list[i] < list[j];
   } else {
      mustSwap = list[i] > list[j];
   }
   …
}
```

Sorting with a configurable order, version B

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

class AscendingComparator  implements Comparator {
  public boolean compare(int i, int j) { return i < j; }
}
class DescendingComparator implements Comparator {
  public boolean compare(int i, int j) { return i > j; }
}

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

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]);
  …
}
```

Which version is better?

Version A:

```
static void sort(int[] list, boolean ascending) {
   …
   boolean mustSwap;
   if (ascending) {
      mustSwap = list[i] < list[j];
   } else {
      mustSwap = list[i] > list[j];
   }
   …
}
```

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?

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, preferably 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

- 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?

A typical Intro CS design process

1. Discuss software that needs to be written
2. Write some code
3. Test the code to identify the defects
4. Debug to find causes of defects
5. Fix the defects
6. If not done, return to step 1

Metrics of software quality

Source: Braude, Bernstein, Software Engineering. Wiley 2011

- Sufficiency / functional correctness
  - Fails to implement the specifications … Satisfies all of the specifications
- Robustness
  - Will crash on any anomalous event … Recovers from all anomalous events
- Flexibility
  - Must be replaced entirely if spec changes … Easily adaptable to changes
- Reusability
  - Cannot be used in another application … Usable without modification
- Efficiency
  - Fails to satisfy speed or storage requirement … satisfies requirements
- Scalability
  - Cannot be used as the basis of a larger version … is basis for much larger version…
- Security
  - Security not accounted for at all … No manner of breaching security is known

Design challenges/goals

Better software design

- Think before coding
- Consider non-functional quality attributes
  - Maintainability, extensibility, performance, …
- Propose, consider design alternatives
  - Make explicit design decisions

Using a design process

- A design process organizes your work
- A design process structures your understanding
- A design process facilitates communication

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

Principles of Software Construction:
Objects, Design, and Concurrency

Part 1: Introduction

Course overview and introduction to software design

Charlie Garrod Bogdan Vasilescu

School of
Computer Science

institute for
SOFTWARE
RESEARCH

Concurrency

- Roughly: doing more than one thing at a time

Summary: Course themes

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

Software Engineering (SE) at CMU

- 17-214: Code-level design
  - Extensibility, reuse, concurrency, functional correctness
- 17-313: Human aspects of software development
  - Requirements, teamwork, scalability, security, scheduling, costs, risks, business models
- 17-413 Practicum, 17-415 Seminar, Internship
- Various courses on requirements, architecture, software analysis, SE for startups, etc.
- SE Minor: http://isri.cmu.edu/education/undergrad

COURSE ORGANIZATION

Preconditions

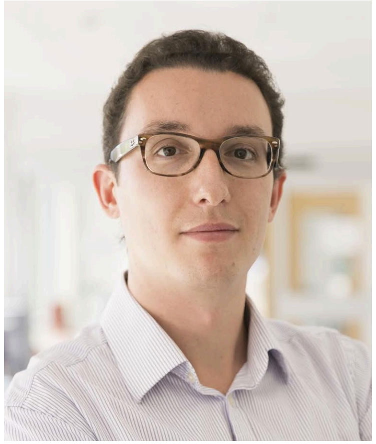
- 15-122 or equivalent
  - Two semesters of programming
  - Knowledge of C-like languages
- 21-127 or equivalent
  - Familiarity with basic discrete math concepts
- Specifically:
  - Basic programming skills
  - Basic (formal) reasoning about programs
    - Pre/post conditions, invariants, formal verification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

Learning goals

- Ability to **design** medium-scale programs
- Understanding **OO programming** concepts & design decisions
- Proficiency with basic **quality assurance** techniques for functional correctness
- Fundamentals of **concurrency**
- Practical skills

Course staff

- Bogdan Vasilescu
  vasilescu@cmu.edu
  Wean 5115



- Charlie Garrod
  charlie@cs.cmu.edu
  Wean 5101

- Teaching assistants: Adithya, Arihant, Bujji, David, Megan, Nick, Tian

Course meetings

Smoking
Section

- Lectures: Tuesday and Thursday 3:00 – 4:20pm DH A302
  - Electronic devices discouraged
- Recitations: Wednesdays 9:30 - … - 2:20pm
  - Supplementary material, hands-on practice, feedback
  - Bring your laptop
- Office hours: see course web page
  - https://www.cs.cmu.edu/~charlie/courses/17-214/

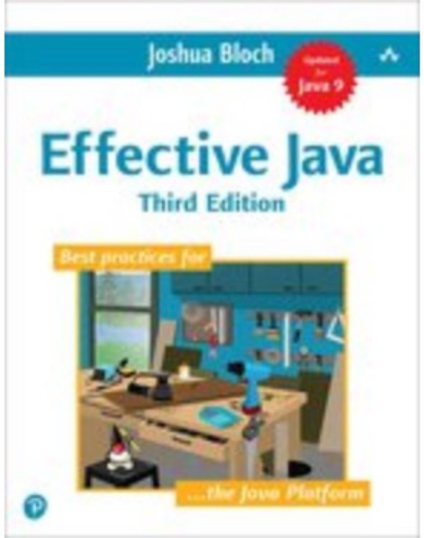
Recitation attendance is required

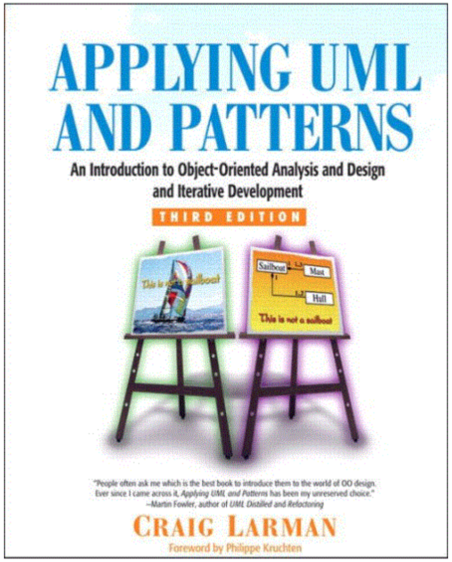
Infrastructure

- Course website: http://www.cs.cmu.edu/~charlie/courses/17-214
  - Schedule, office hours calendar, lecture slides, policy documents
- Tools
  - Git, Github: Assignment distribution, hand-in, and grades
  - Piazza: Discussion board
  - Eclipse or IntelliJ: Recommended for code development (other IDEs are fine)
  - Gradle, Travis-CI, Checkstyle, Findbugs: Practical development tools
- Assignments
  - Homework 1 available tomorrow
- First recitation is tomorrow
  - Introduction to Java and the tools in the course
  - Install Git, Java, some IDE, Gradle beforehand

Textbooks

- Required course textbooks (electronically available through CMU library):
  – Joshua Bloch. Effective Java, Third Edition. Addison-Wesley, ISBN 978-0-13-468599-1.
  – Craig Larman. Applying UML and Patterns. 3rd Edition. Prentice Hall, ISBN 978-0321356680.
- Additional readings on design, Java, and concurrency on the course 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 (also see the course syllabus)

- *We expect your work to be your own*
  - You must clearly cite external resources so that we can evaluate your own personal contributions.
- Do not release your solutions (not even after end of semester)
- Ask if you have any questions
- If you are feeling desperate, please mail/call/talk to us
  - Always turn in any work you've completed *before* the deadline
- We use cheating detection tools

Late day policy

- You may turn in each* homework up to 2 days late
- You have five free late days per semester
  - 10% penalty per day after free late days are used
- We don't accept work 3 days late
- See the syllabus for additional details
- Got extreme circumstances? Talk to us

10% quizzes and participation

- Recitation participation counts toward your participation grade
- Lecture has in-class quizzes

Summary

- Software engineering requires decisions, judgment
- Good design follows a process
- You will get lots of practice in 17-214!

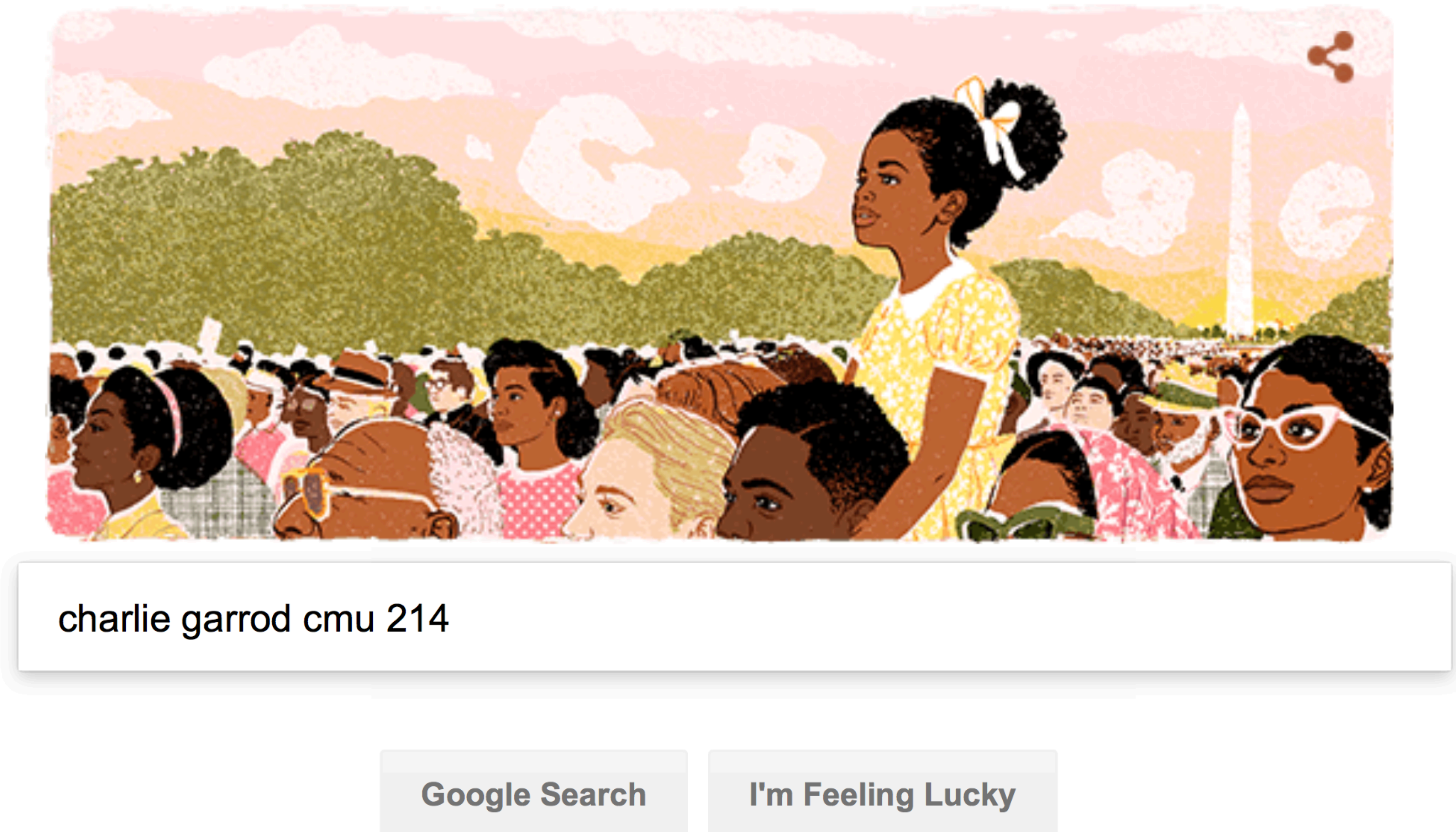
Principles of Software Construction: Objects, Design, and Concurrency

Introduction to course infrastructure

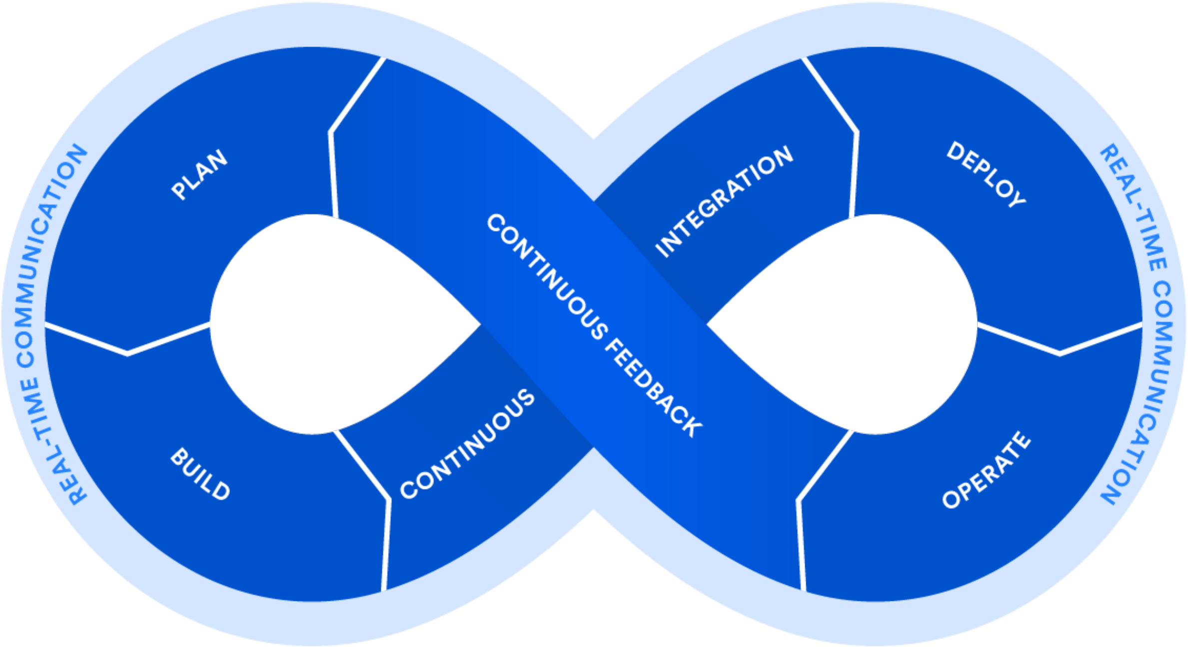
Charlie Garrod **Bogdan Vasilescu**

School of Computer Science

Remember: class website

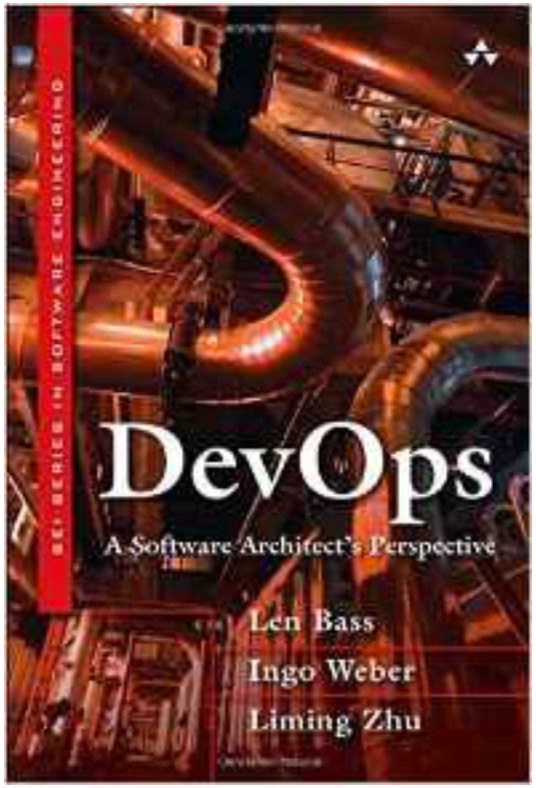


DevOps

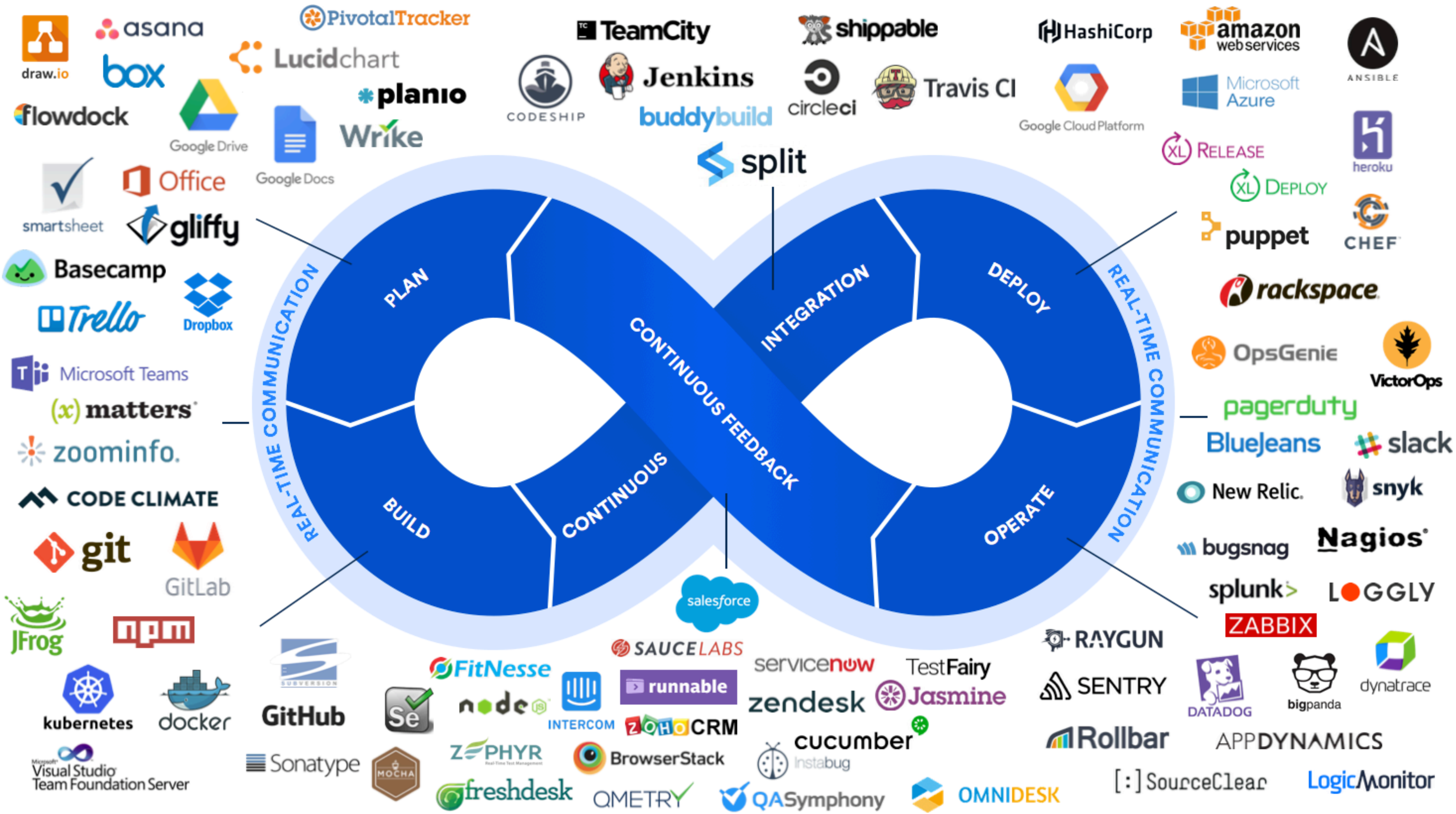

REAL-TIME COMMUNICATION
PLAN
BUILD
CONTINUOUS
CONTINUOUS FEEDBACK
INTEGRATION
DEPLOY
OPERATE
REAL-TIME COMMUNICATION

A DevOps Definition

- "DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality."



DevOps Toolchain



https://marketplace-cdn.atlassian.com/s/f01dfe0a9e6d2f8a1d1bada432a8914f126aea8b/public/devops-hero.png

You will need for homework 1

- Java (+Eclipse/IntelliJ): 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	…

Brief timeline of VCS

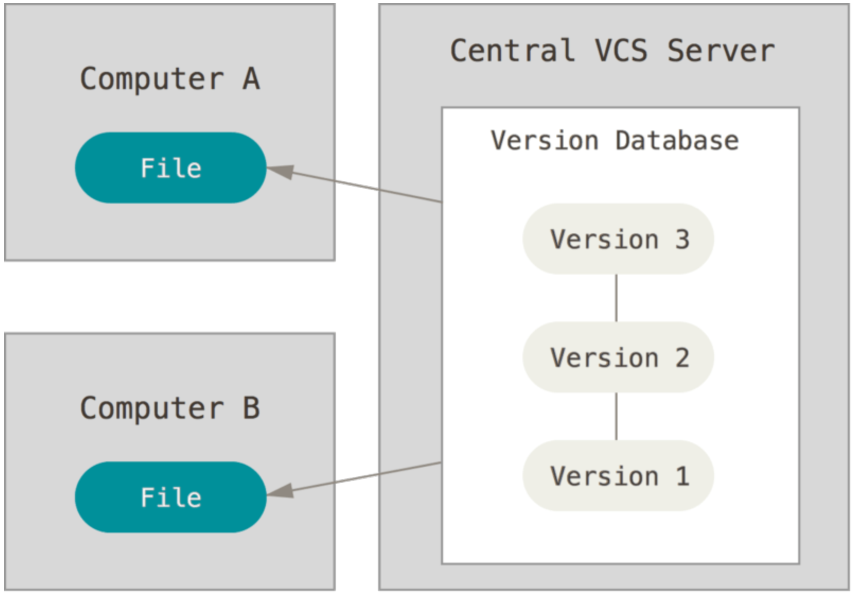
- 1982: RCS (Revision Control System), still maintained
- 1990: CVS (Concurrent Versions System)
- 2000: SVN (Subversion)
- 2005: Bazaar, Git, Mercurial

Git

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

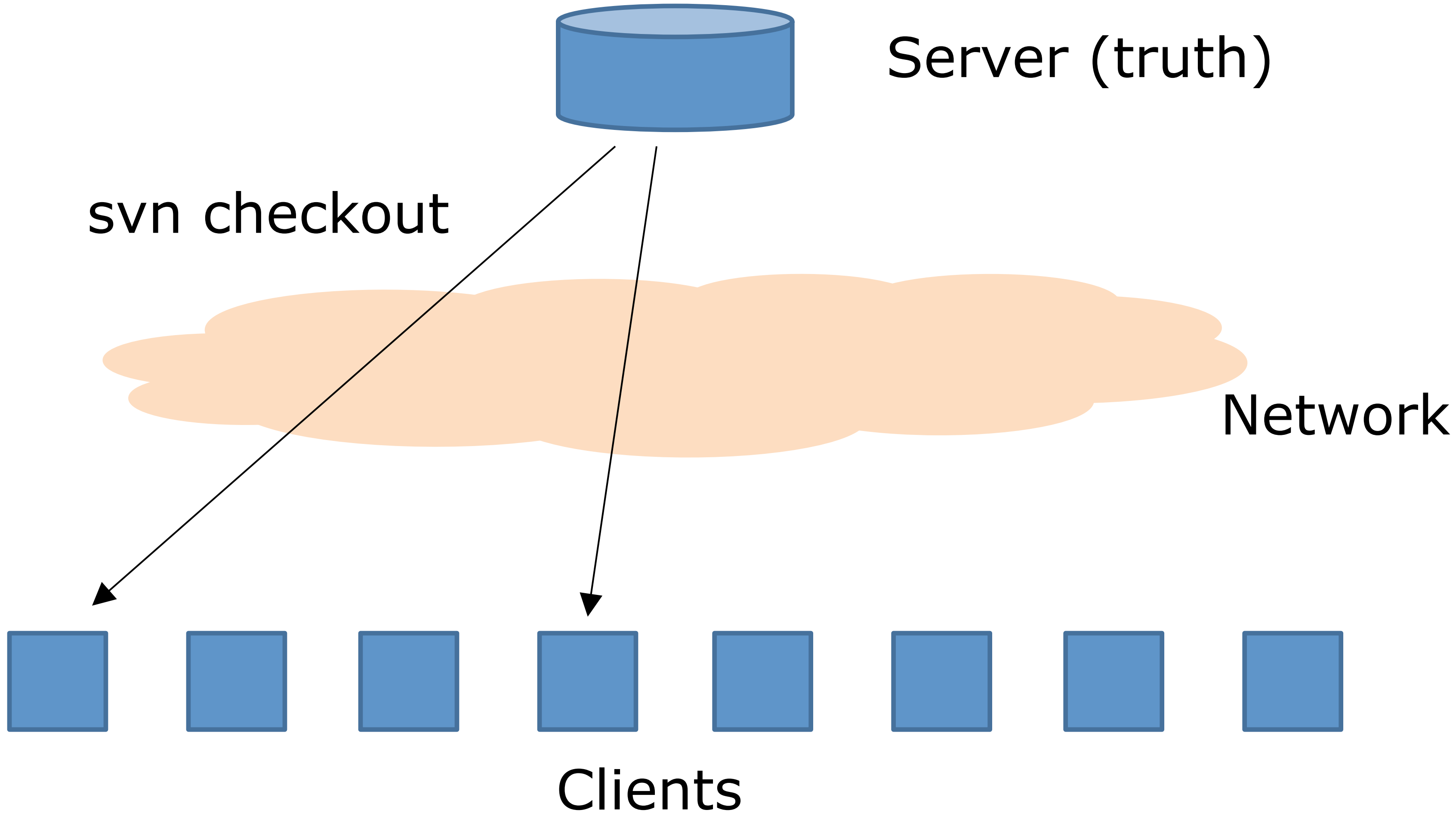
Centralized version control

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

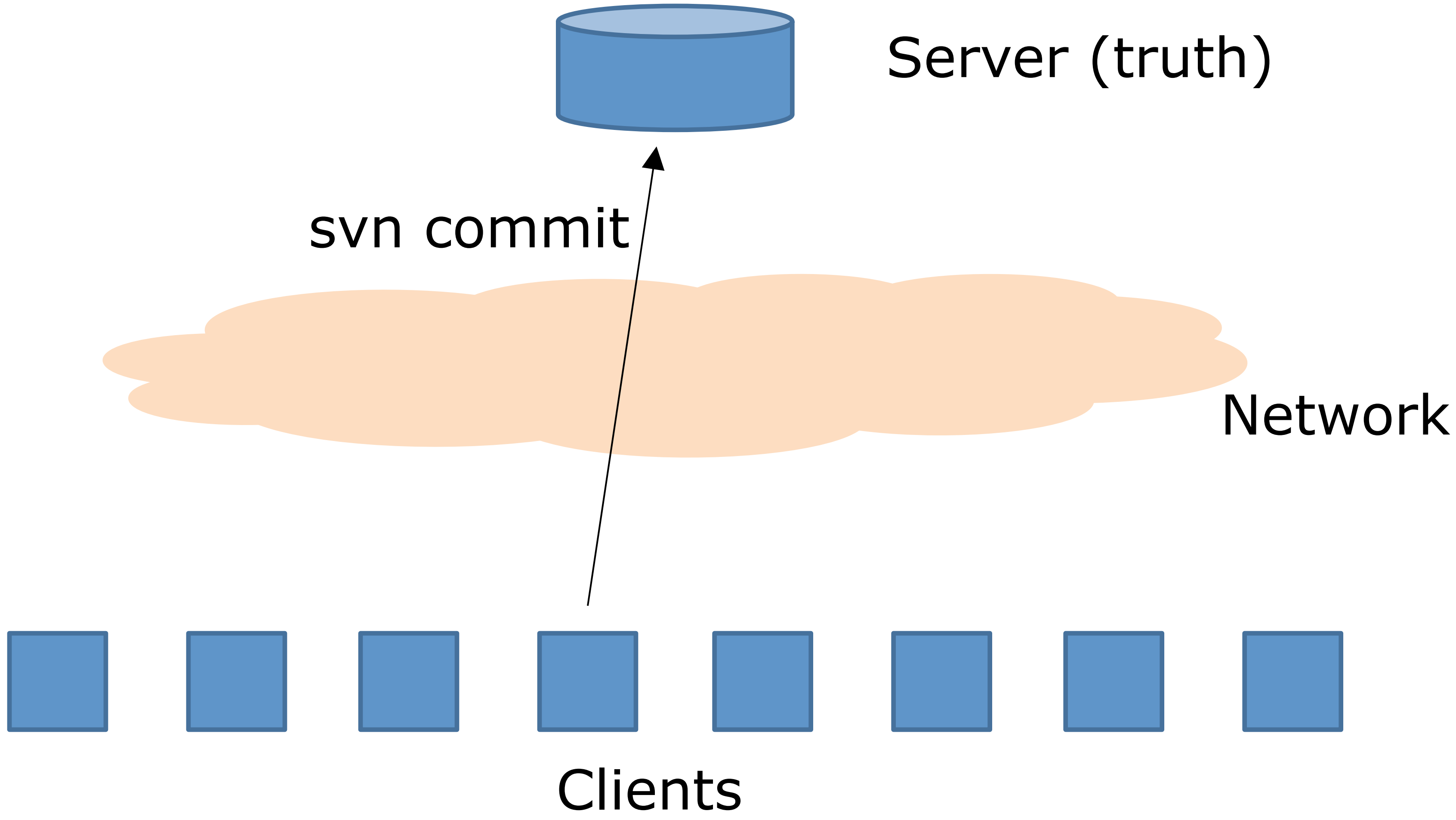


https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

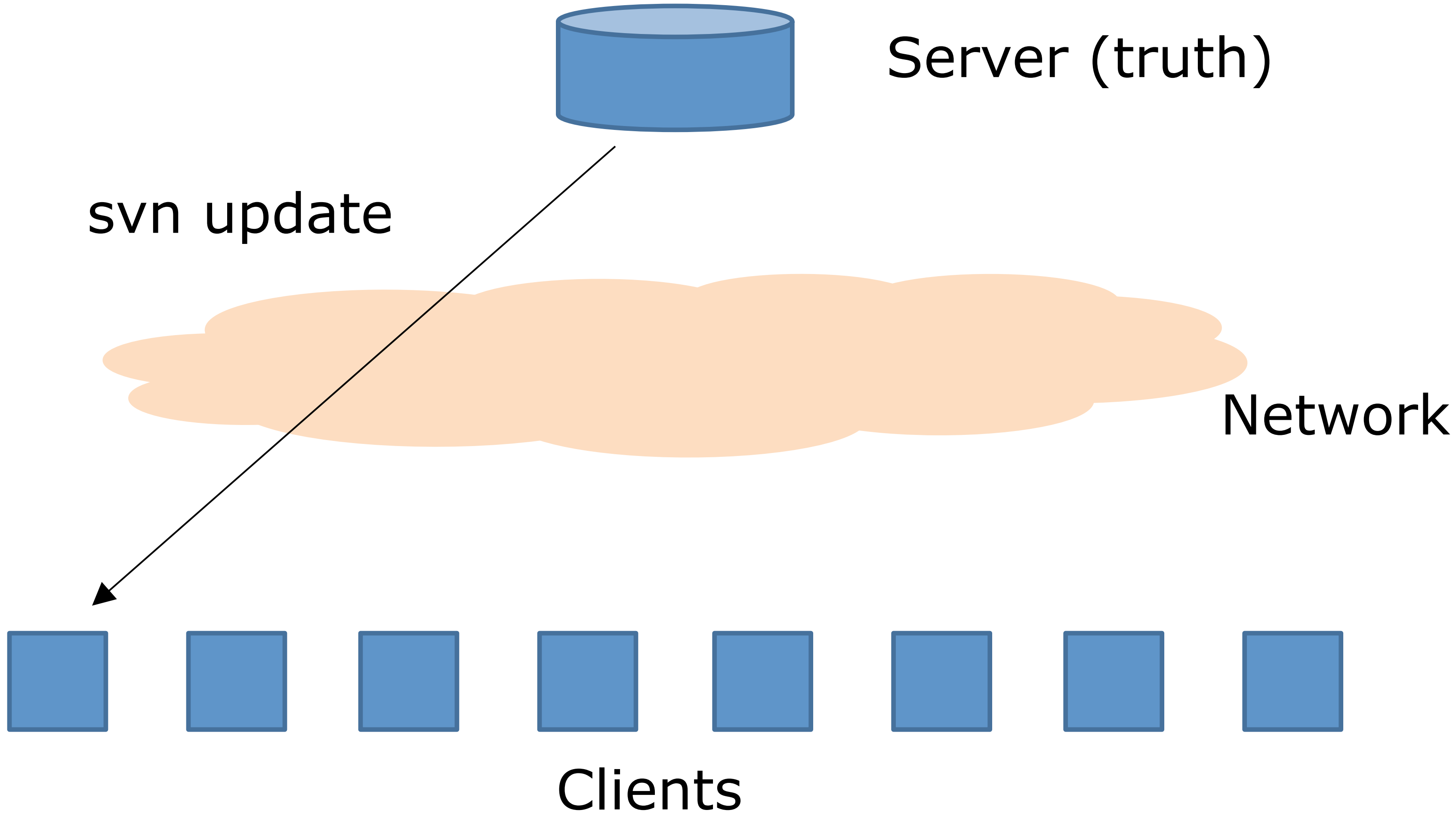
SVN



SVN

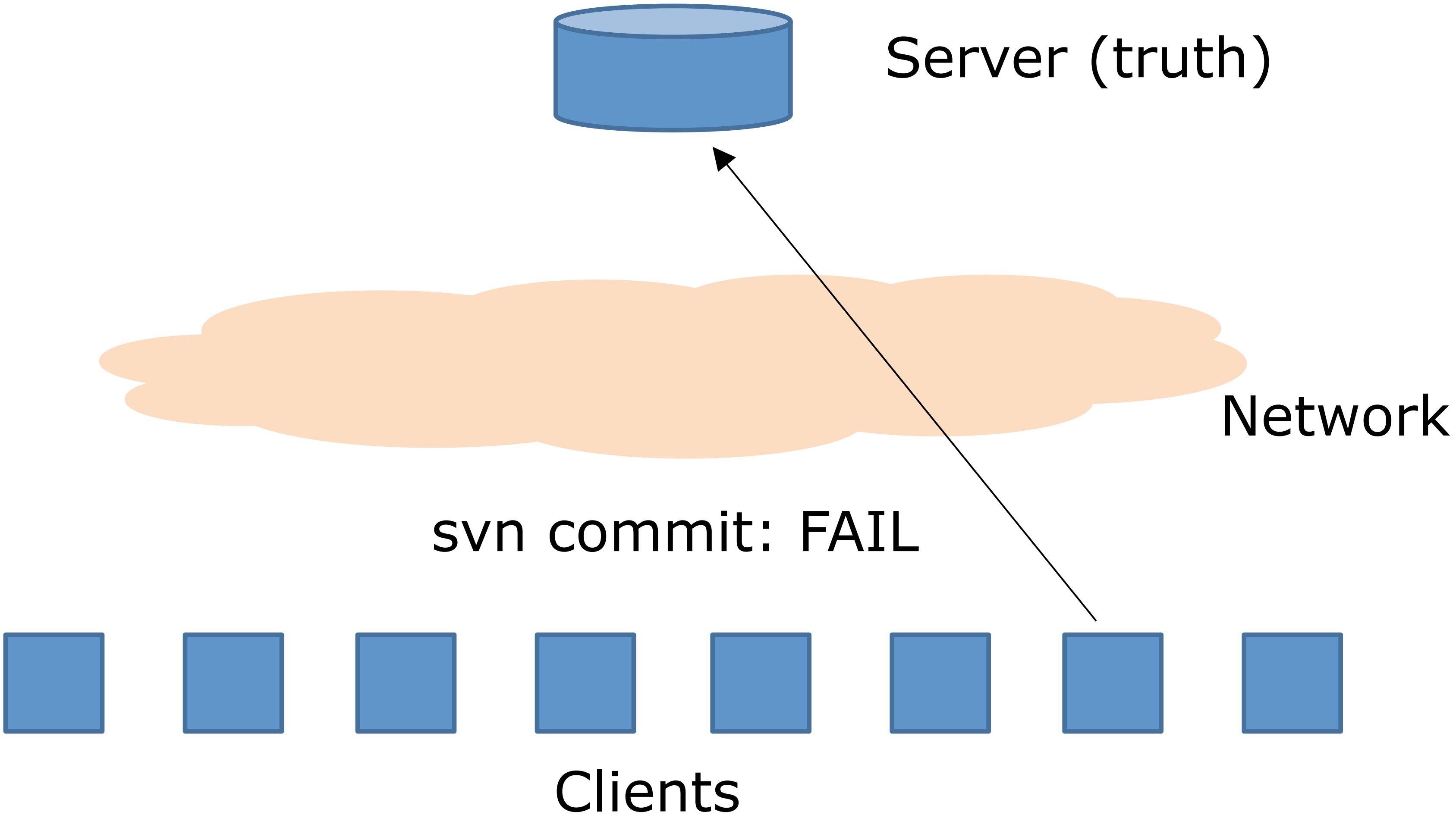


SVN

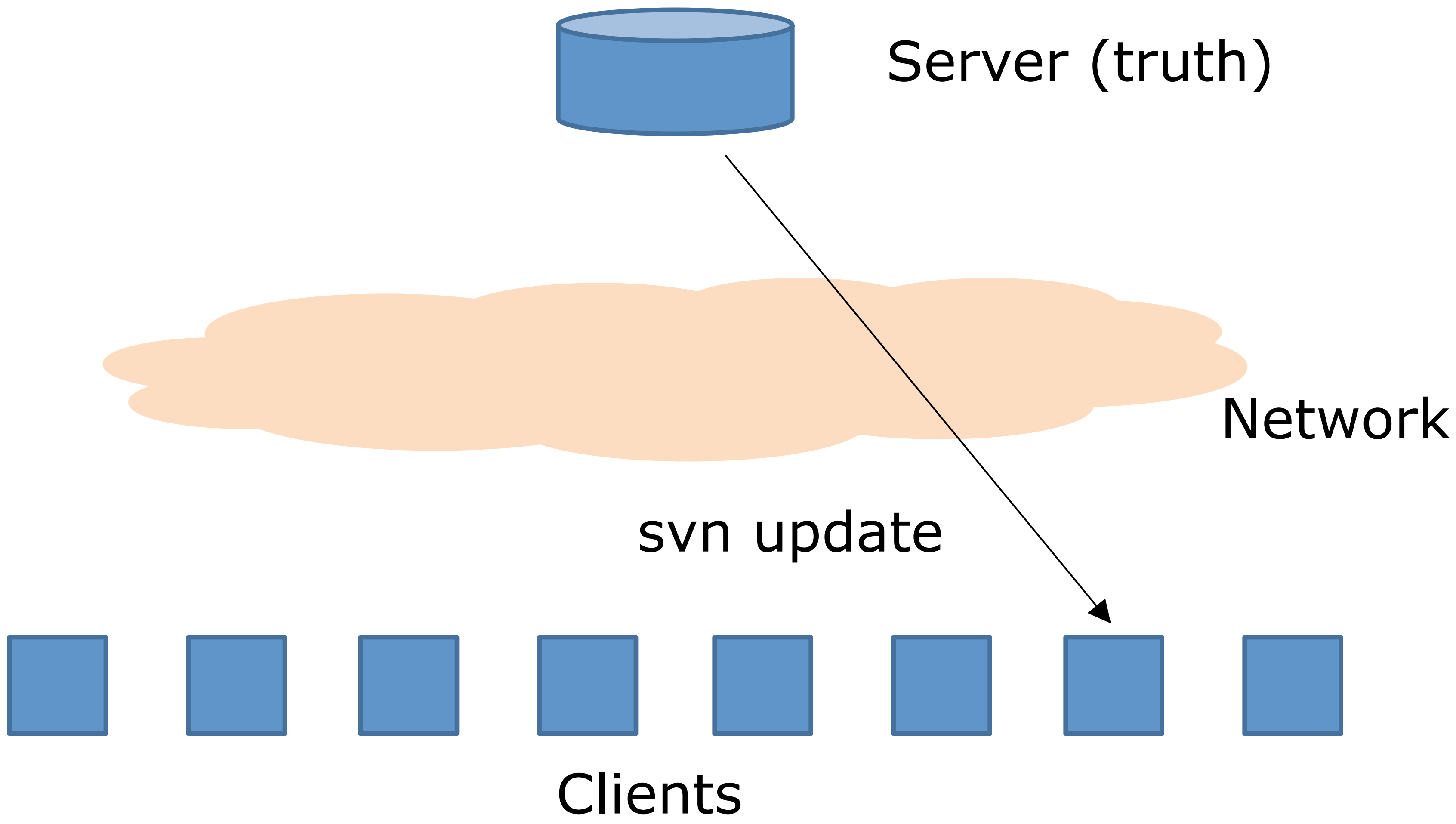


institute for SOFTWARE RESEARCH

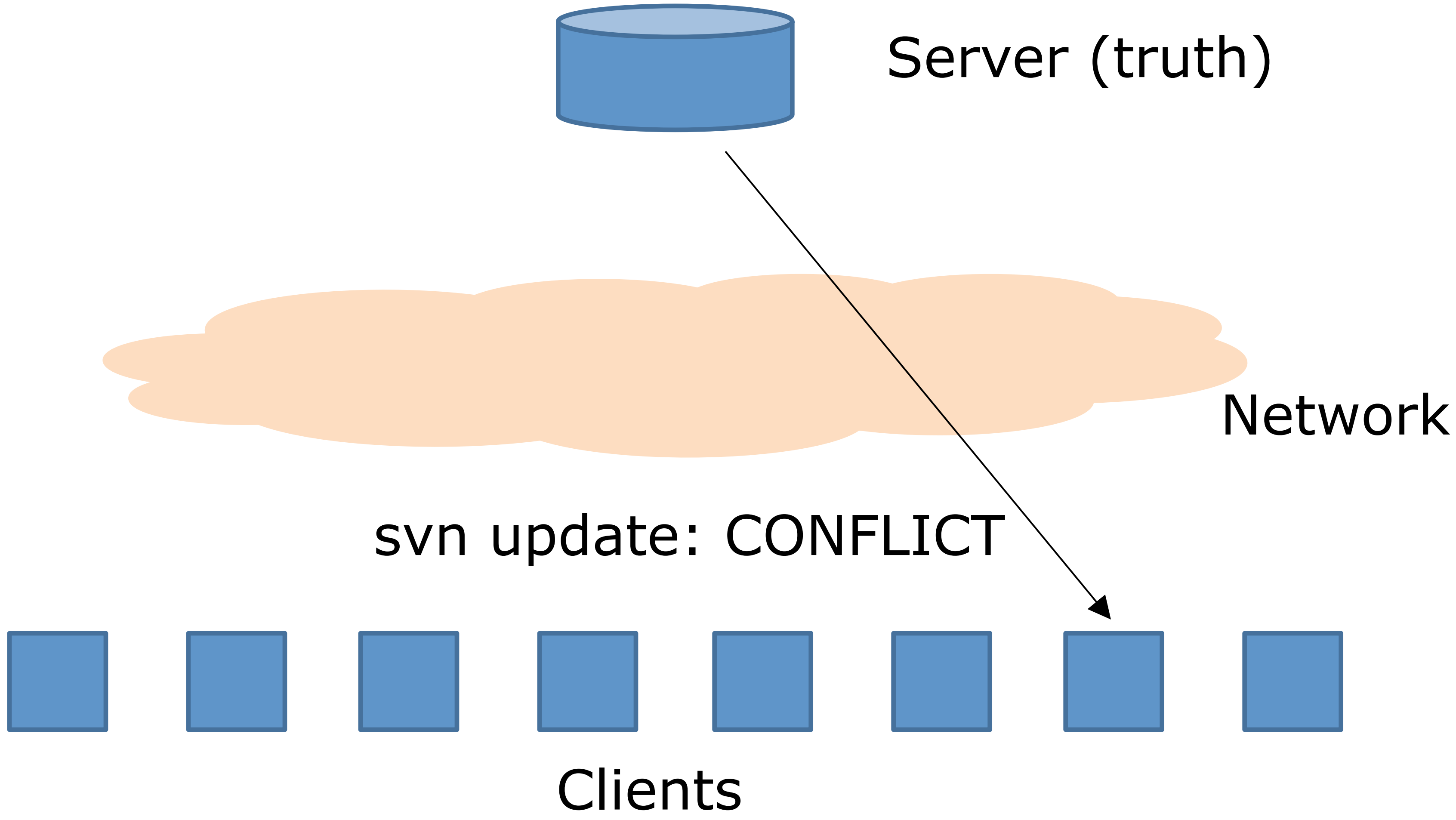
SVN



SVN



SVN



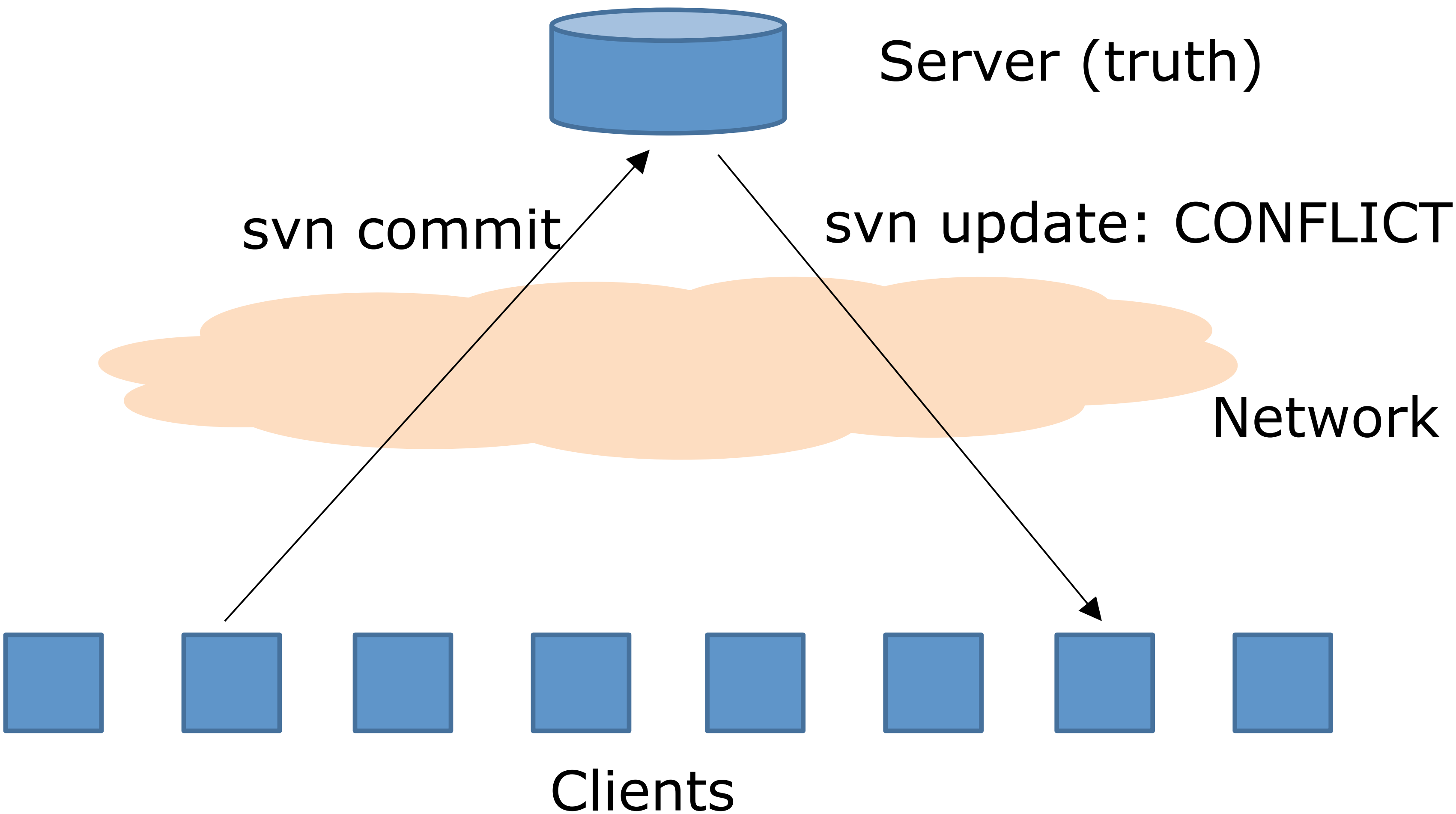
Centralized version control

- 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

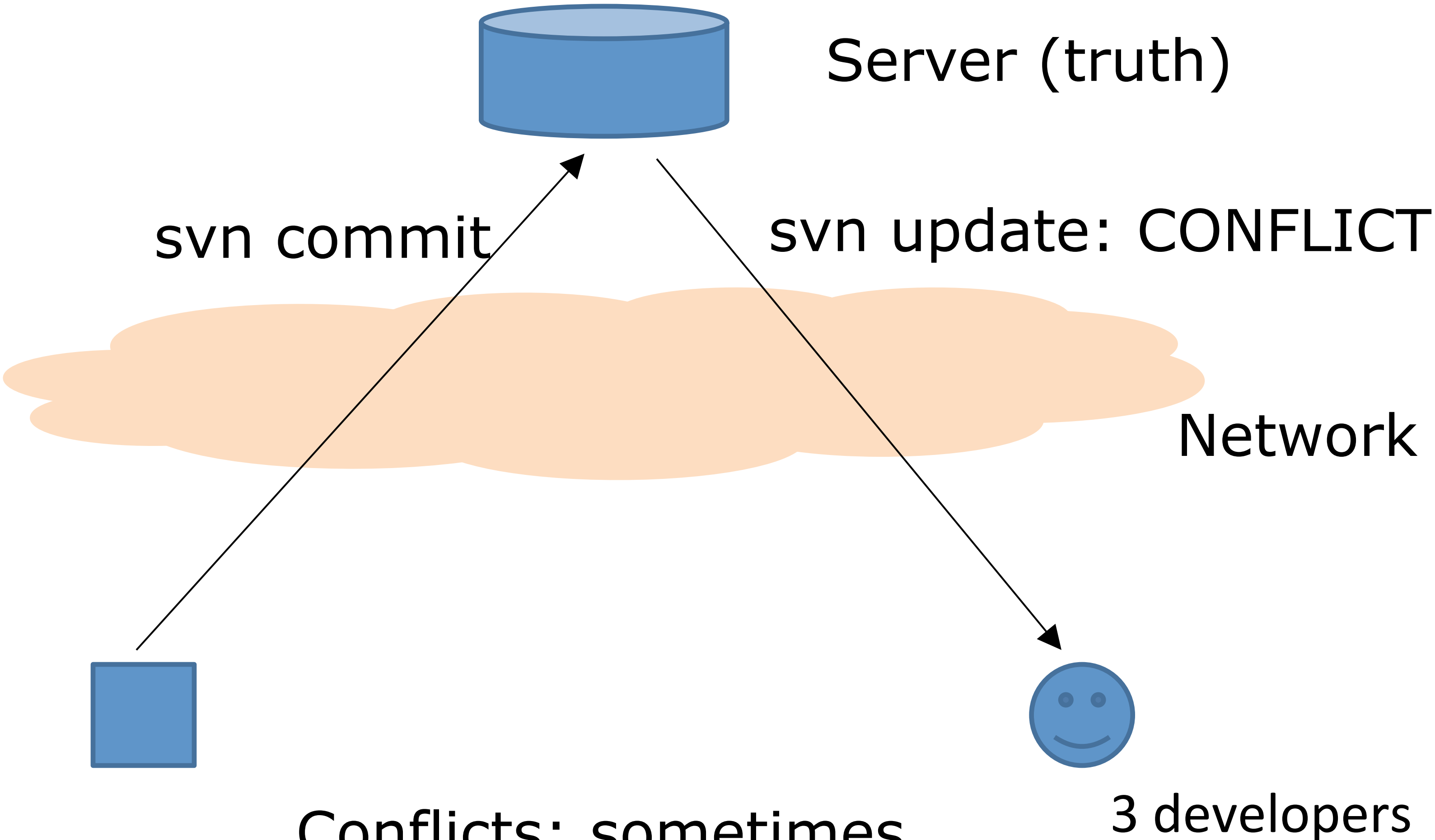
- Easier to lose data
- Incentive to use version control sparingly
- Tangled instead of atomic commits

SVN

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

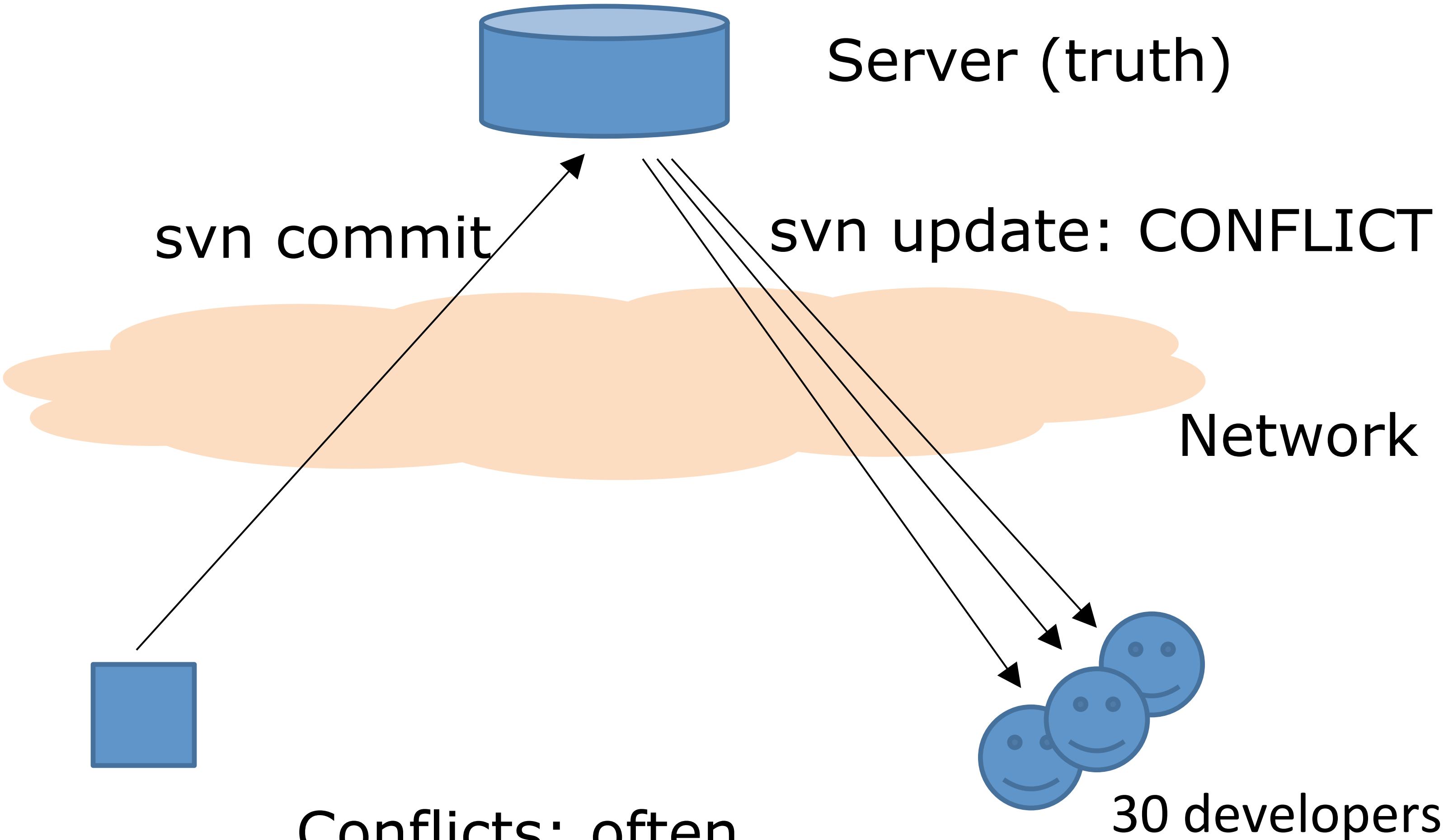


SVN



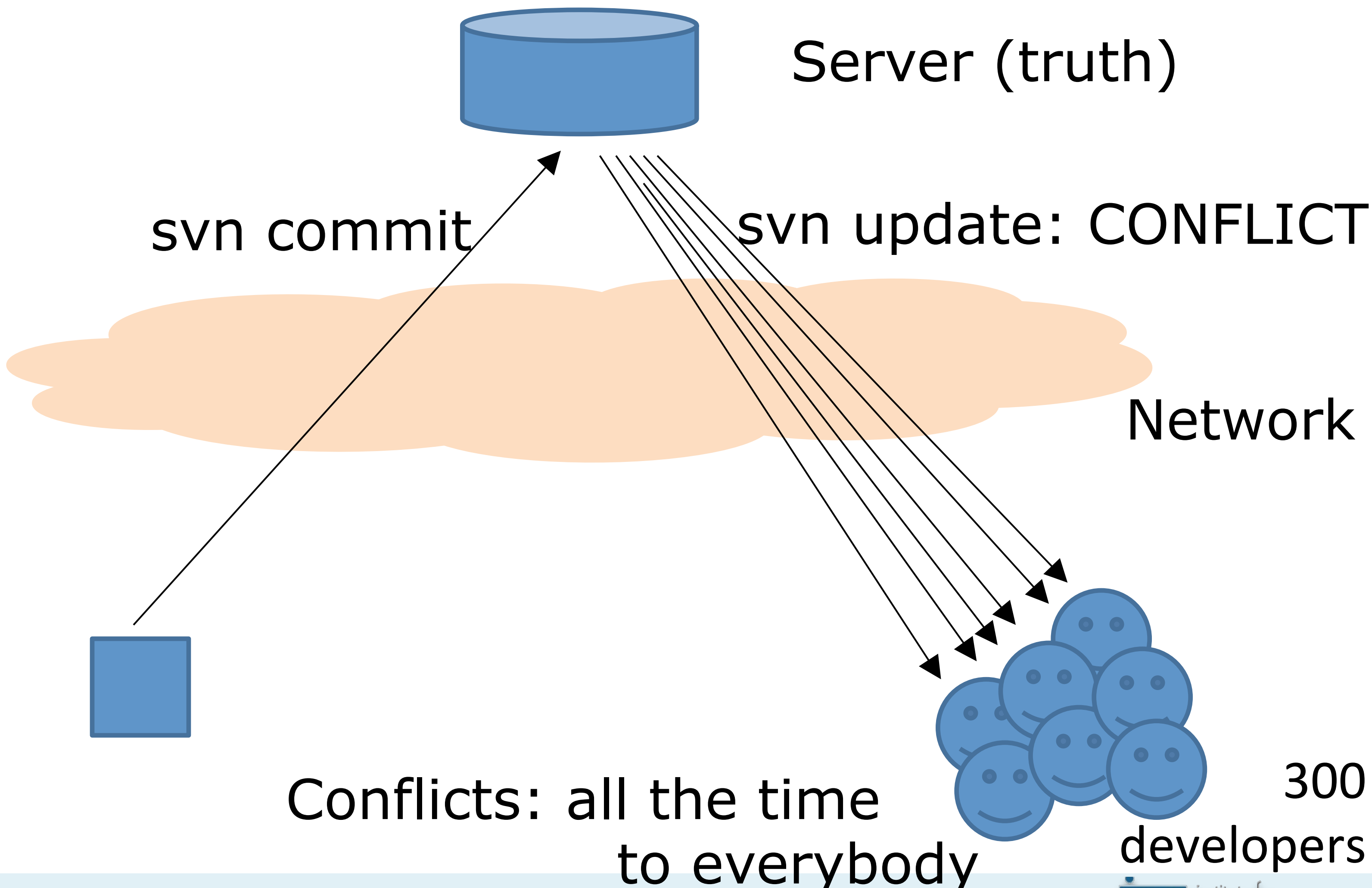
Conflicts: sometimes

SVN



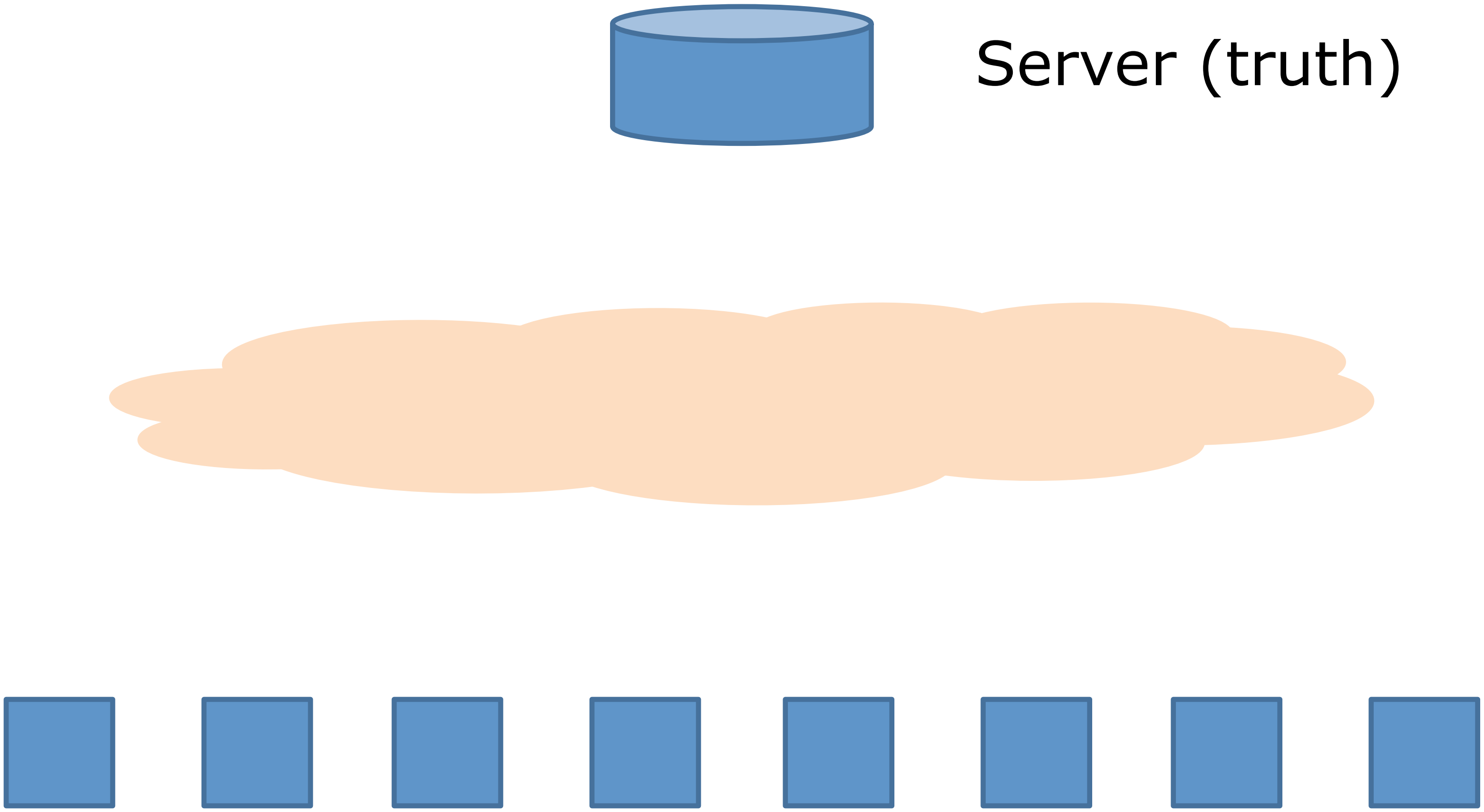
Conflicts: often

SVN


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

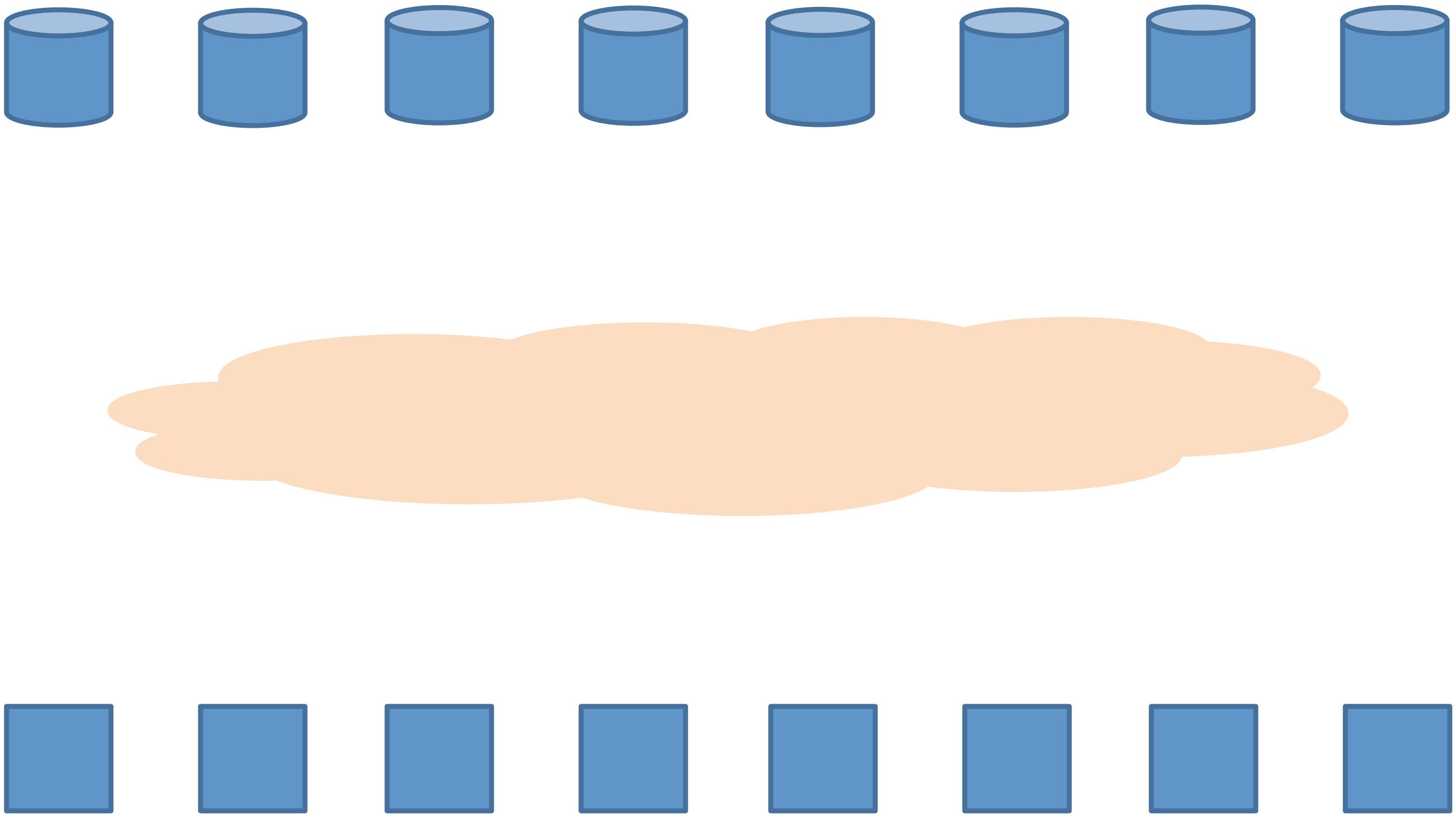
Git is distributed. There is not one server …

Git



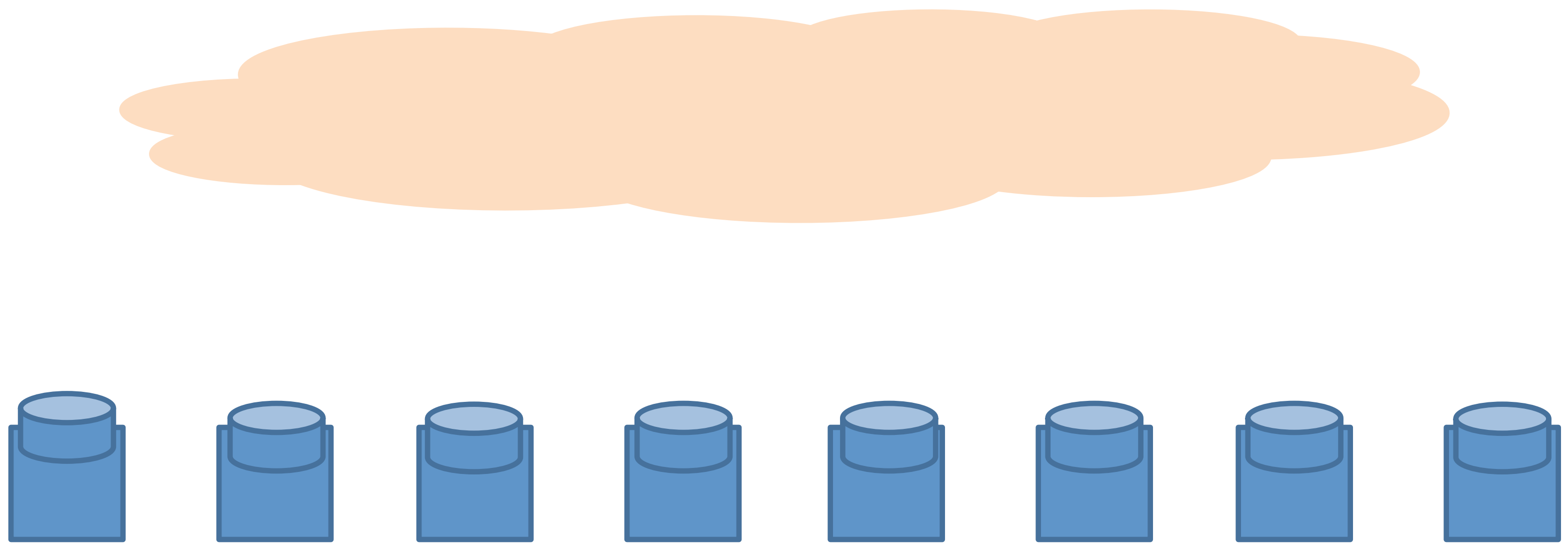
… but many

Git



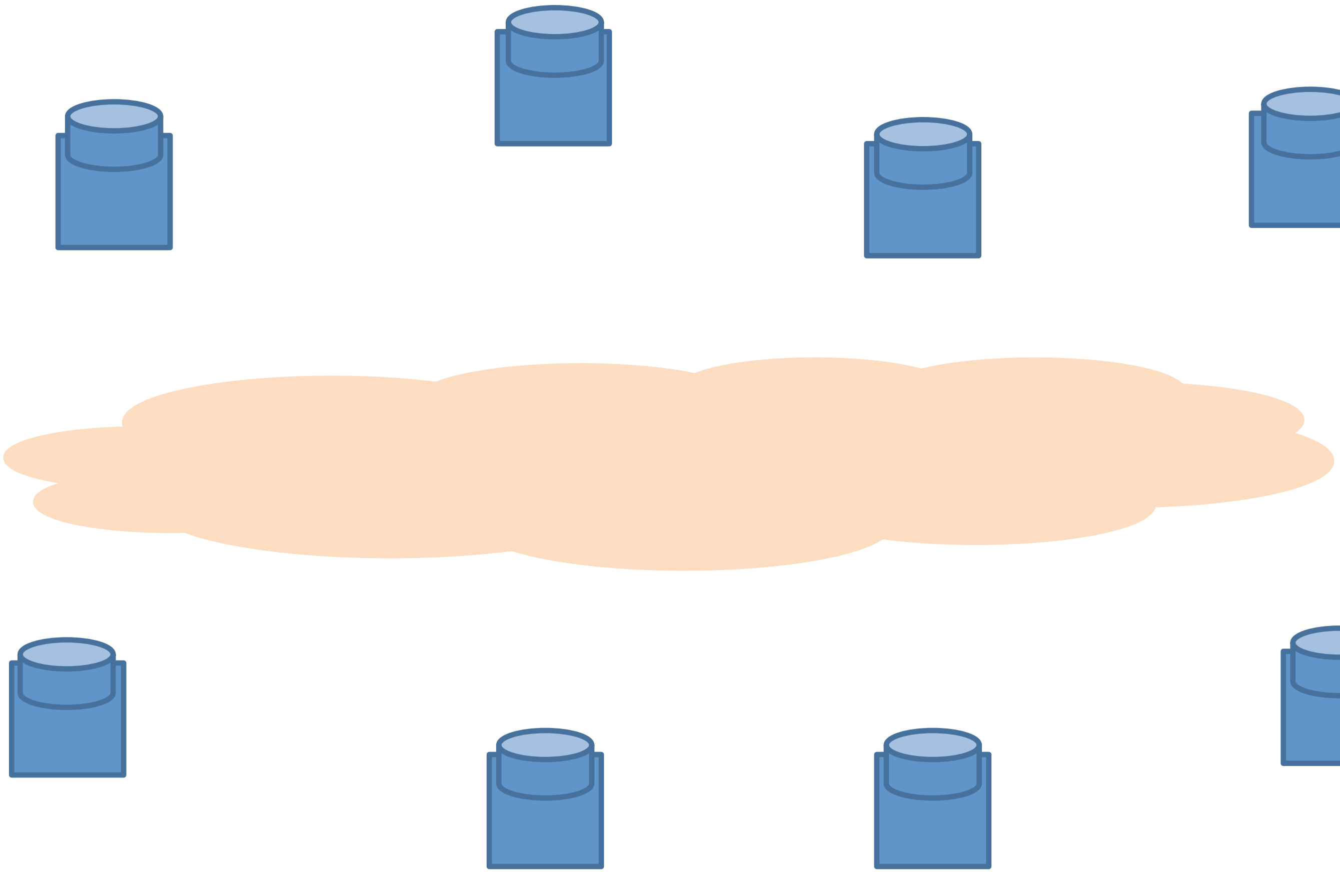
Actually there is one server per computer

Git



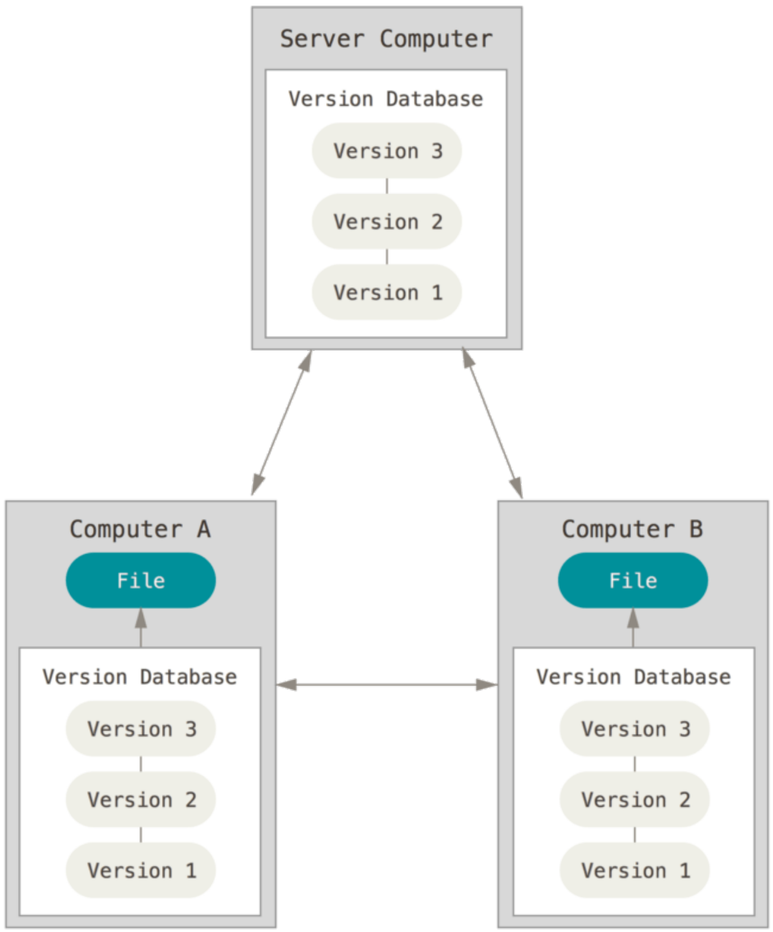
Git

Every computer is a server and version control happens locally.



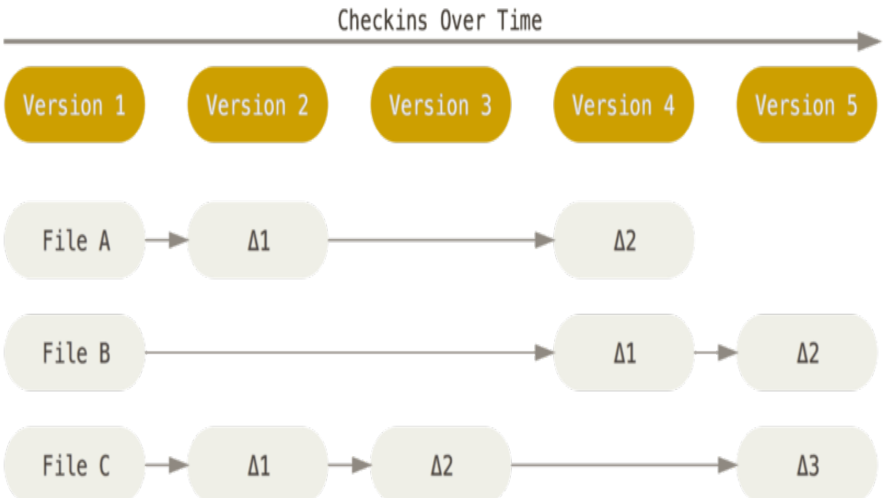
Distributed version control

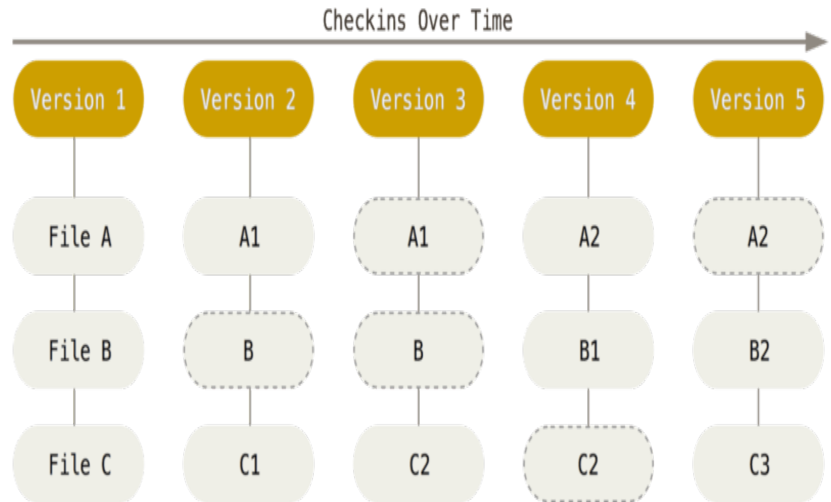
- Clients fully mirror the repository
  - Every clone is a full backup of ***all*** the data
- Advantages:
  - Fast, works offline, scales
  - Better suited for collaborative workflows
- E.g., Git, Mercurial, Bazaar



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

SVN (left) vs. Git (right)





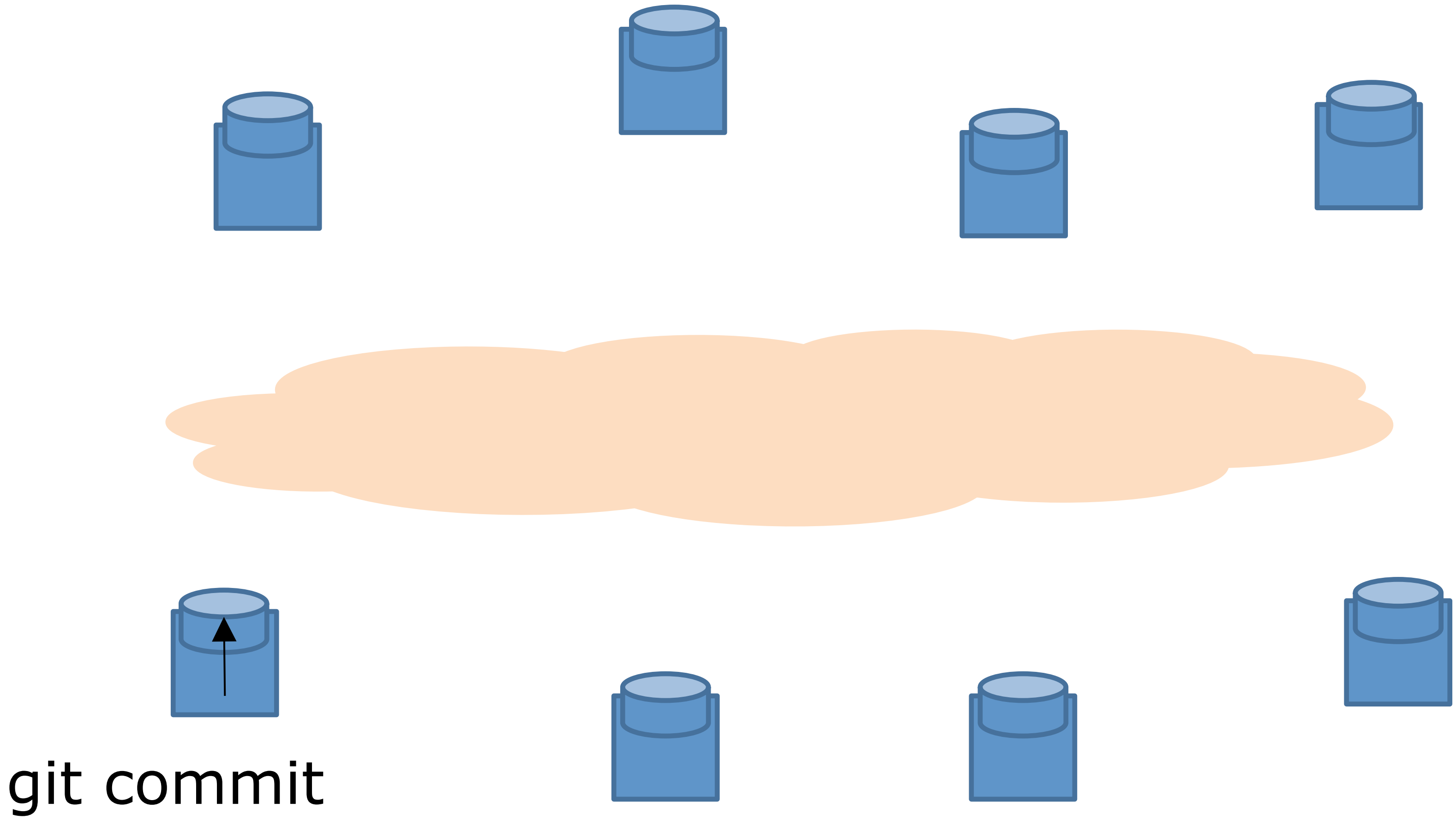
- SVN stores changes to a base version of each file
- Version numbers (1, 2, 3, …) are increased by one after each commit

- 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

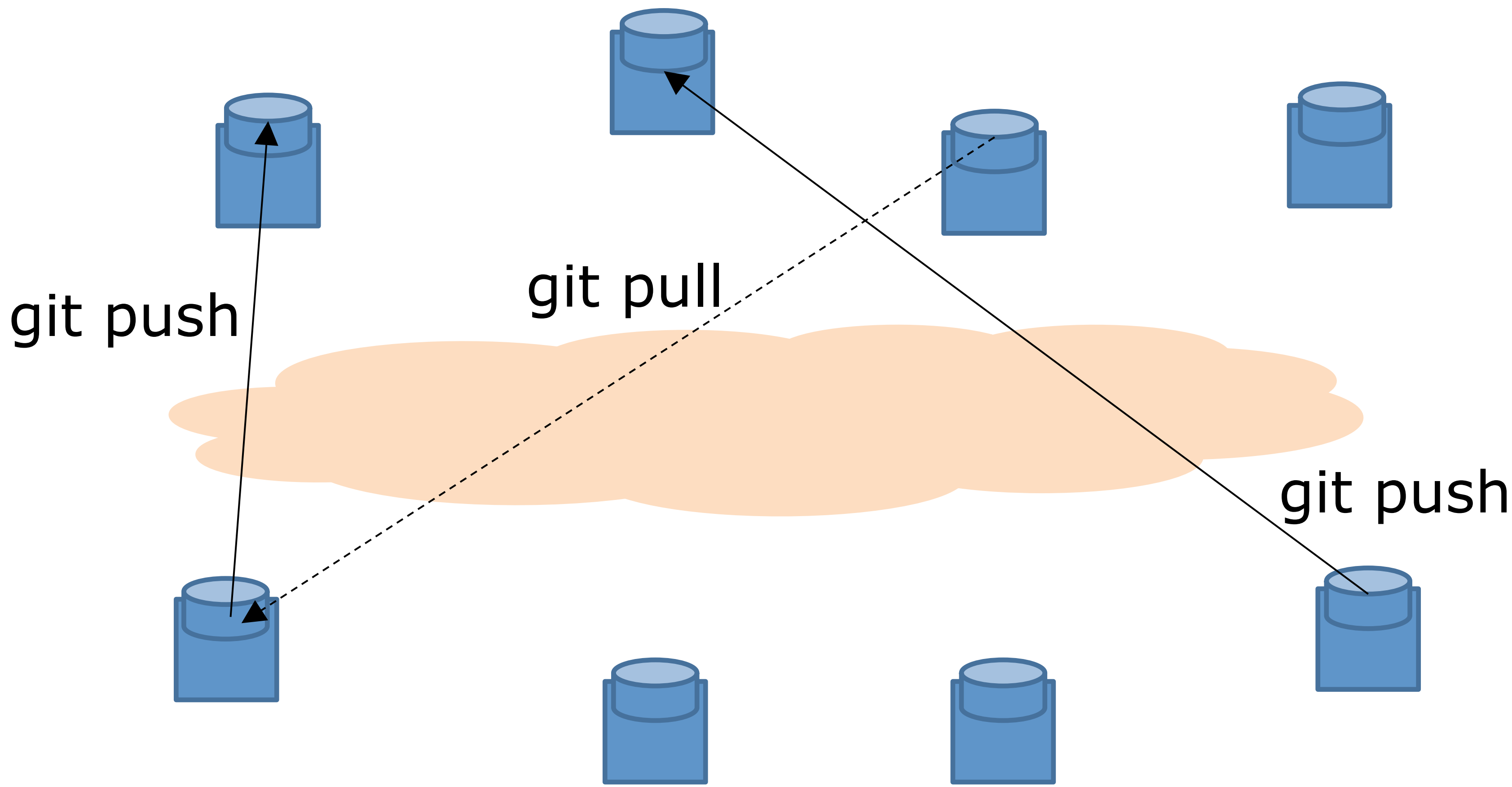
Git

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



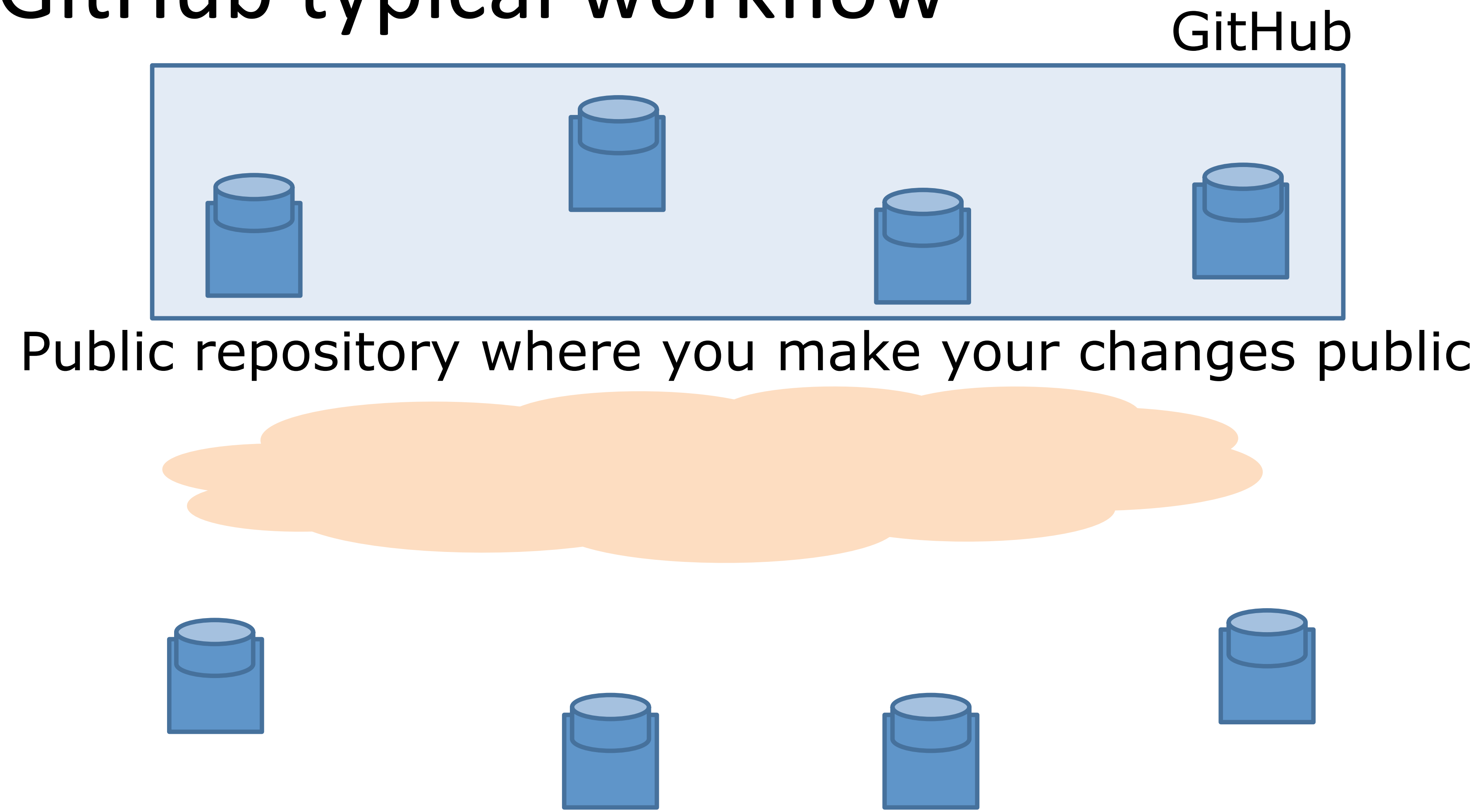
Git

You *push* your commits into their repositories / They *pull* your commits into their repositories

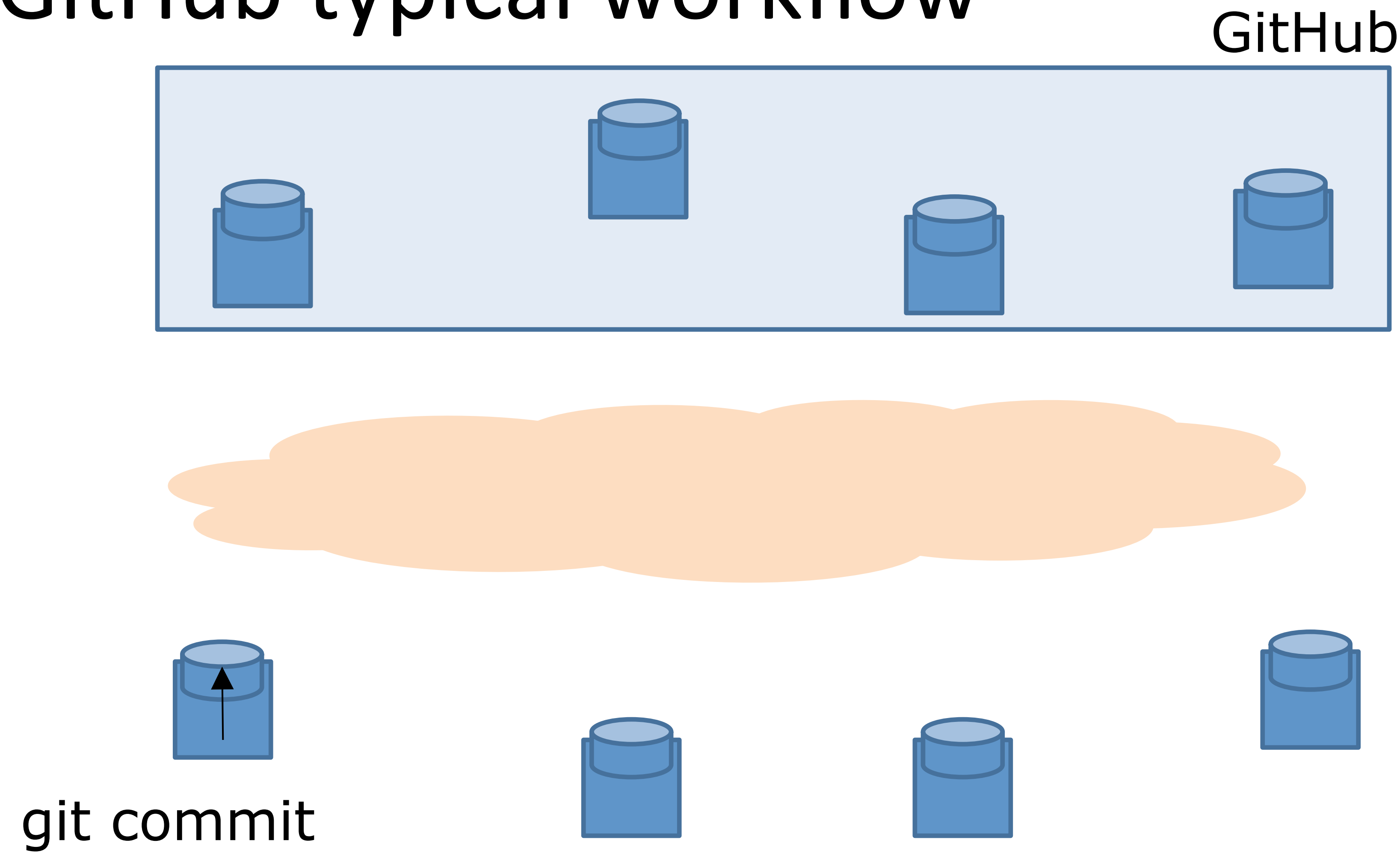


… But requires host names / IP addresses

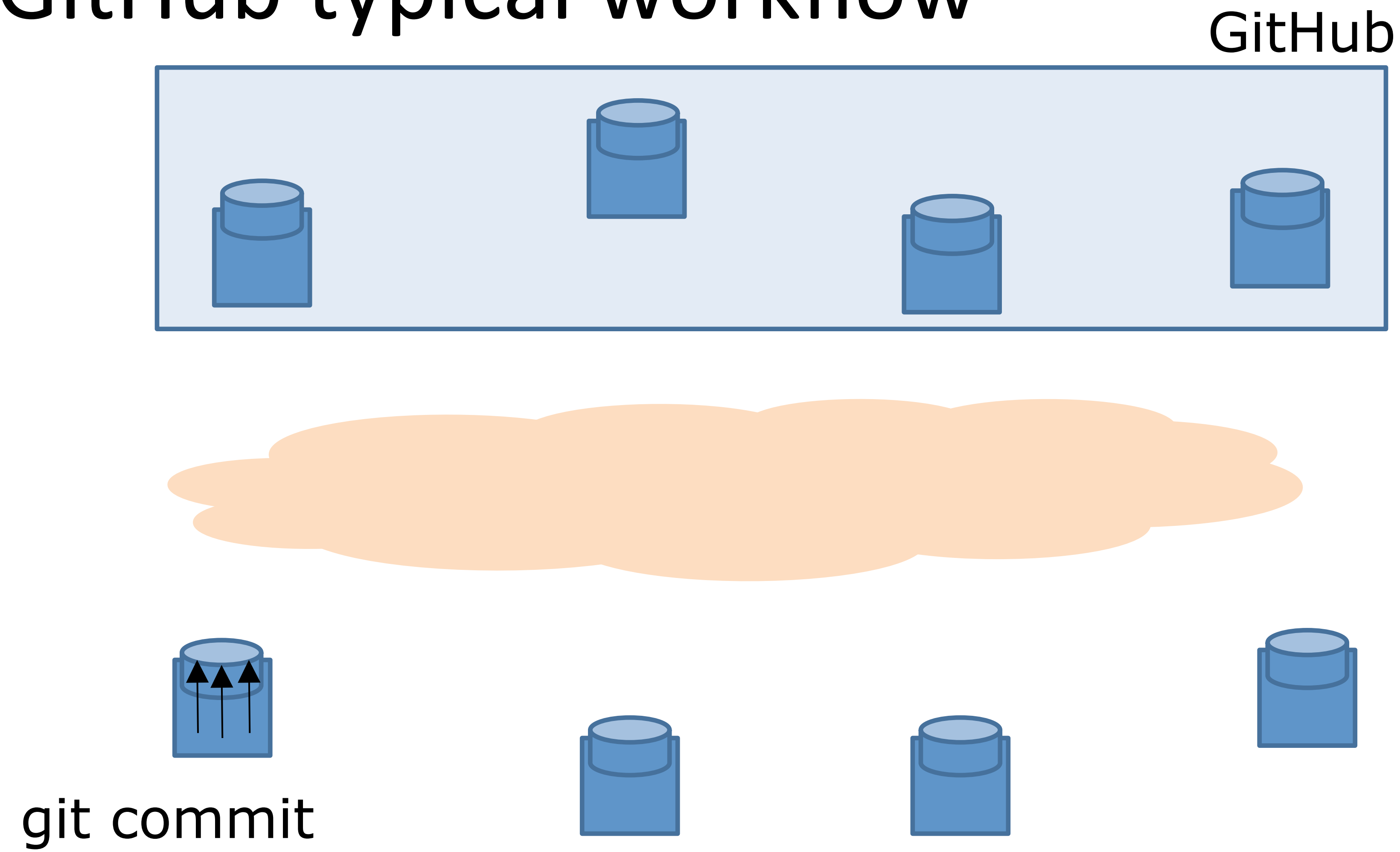
GitHub typical workflow



GitHub typical workflow

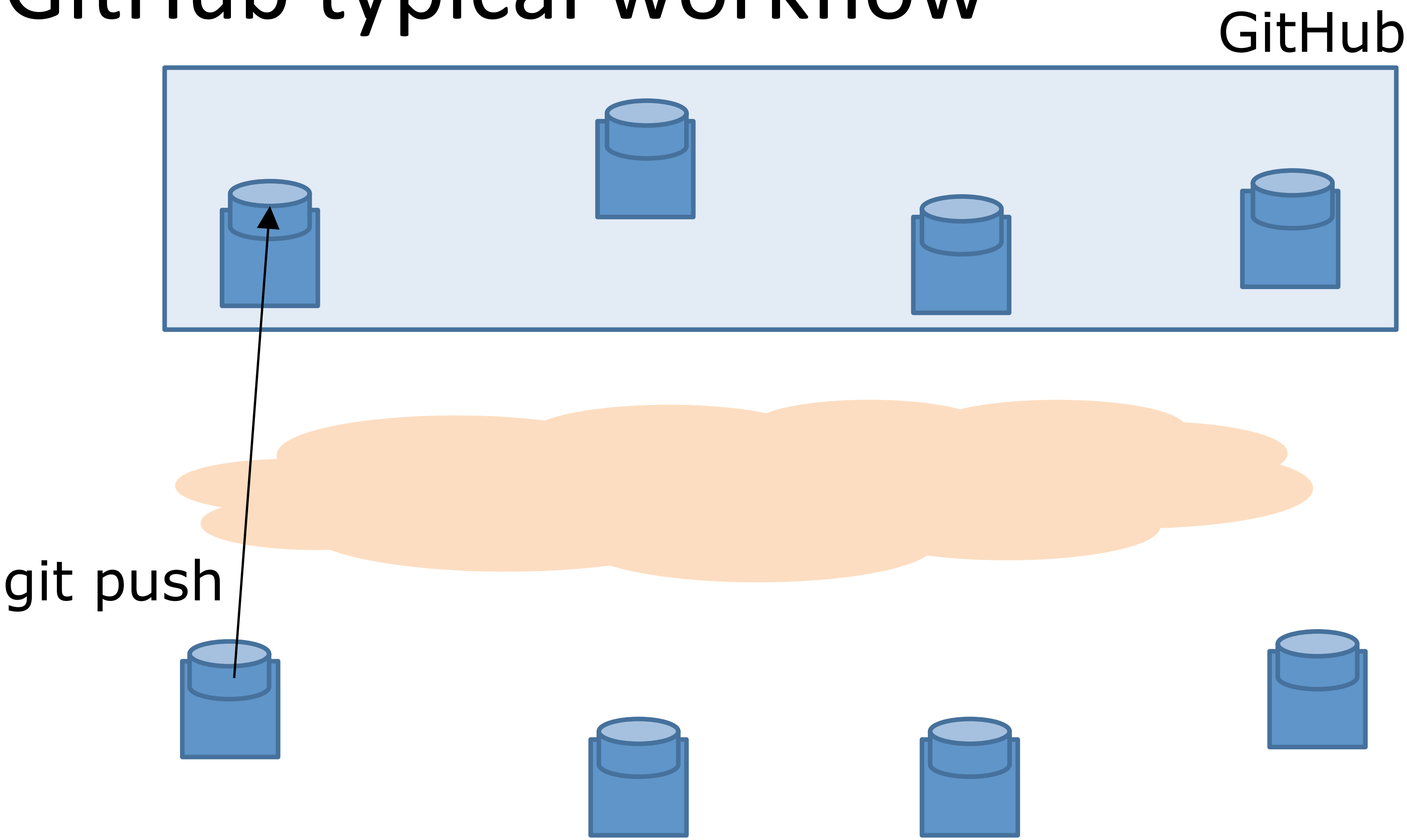


GitHub typical workflow



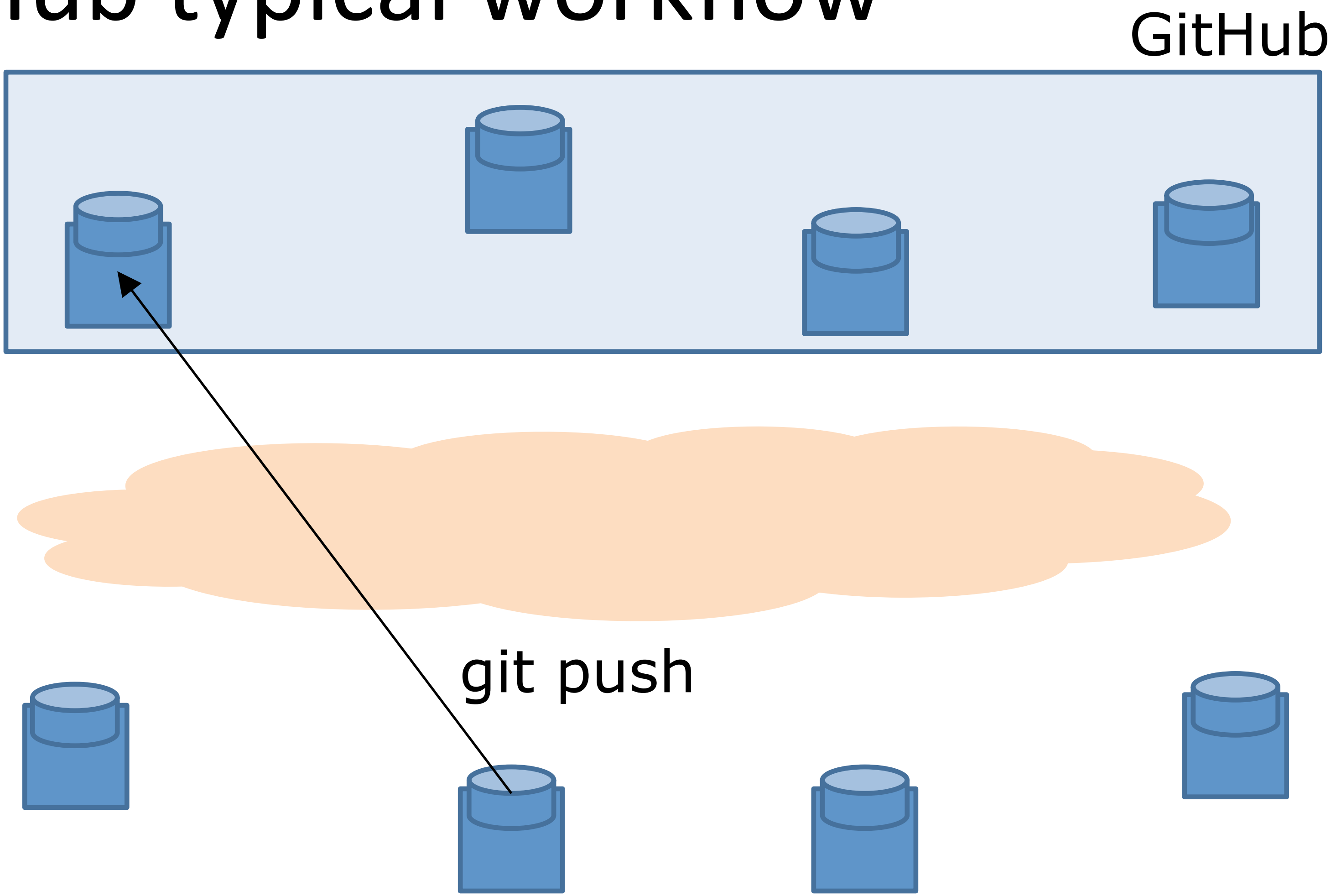
institute for SOFTWARE RESEARCH

GitHub typical workflow



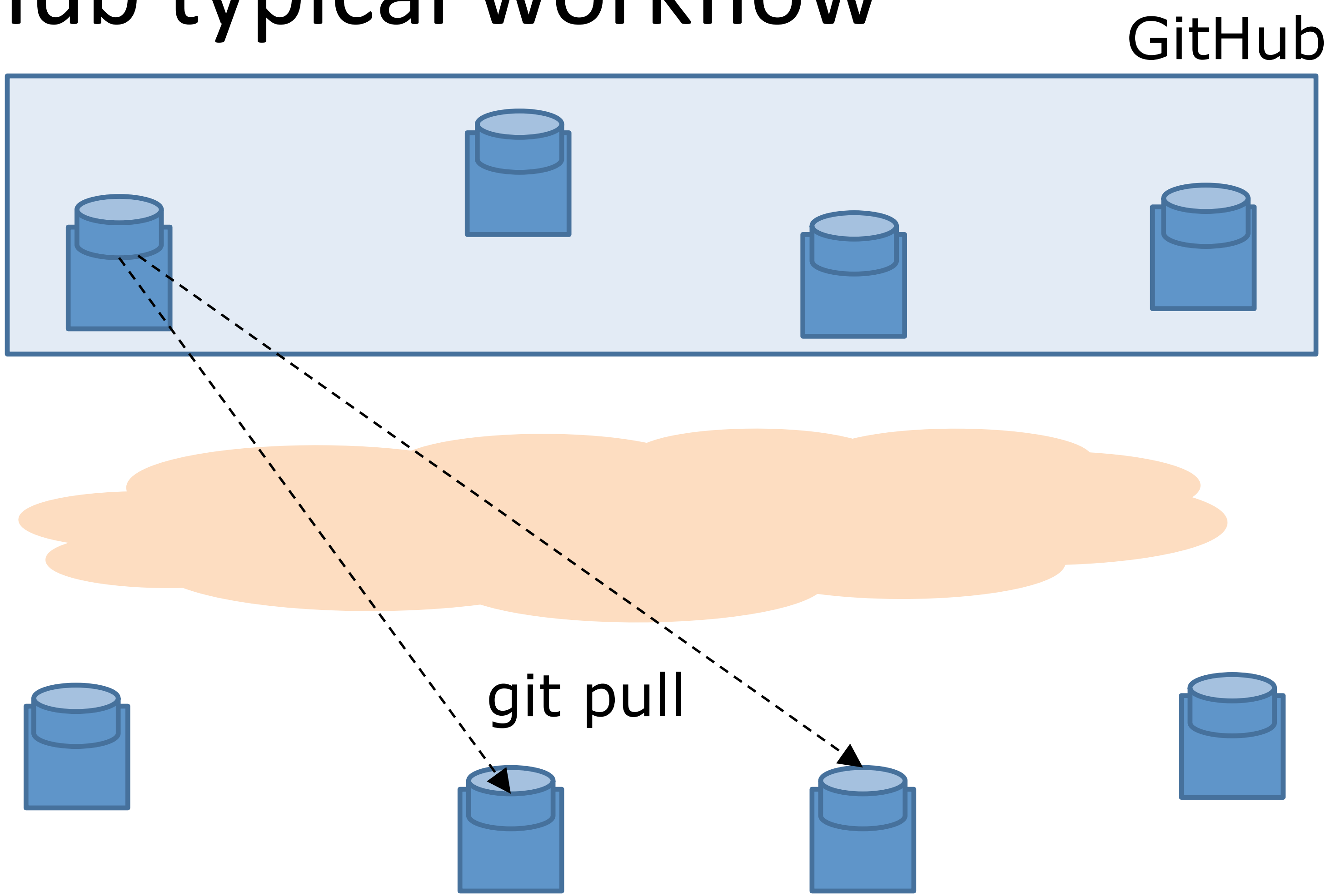
push your local changes into a remote repository.

GitHub typical workflow



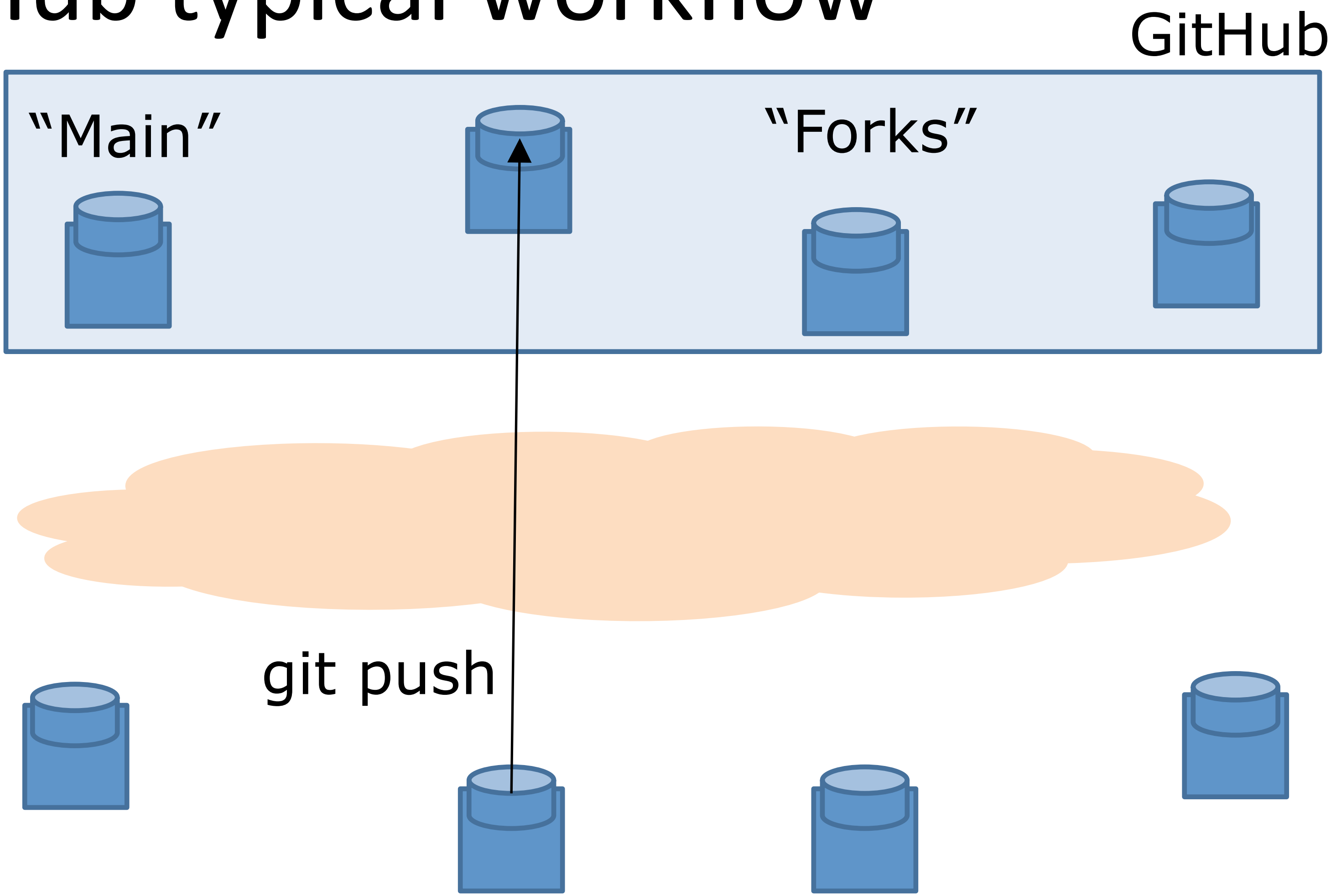
Collaborators can push too if they have access rights.

GitHub typical workflow



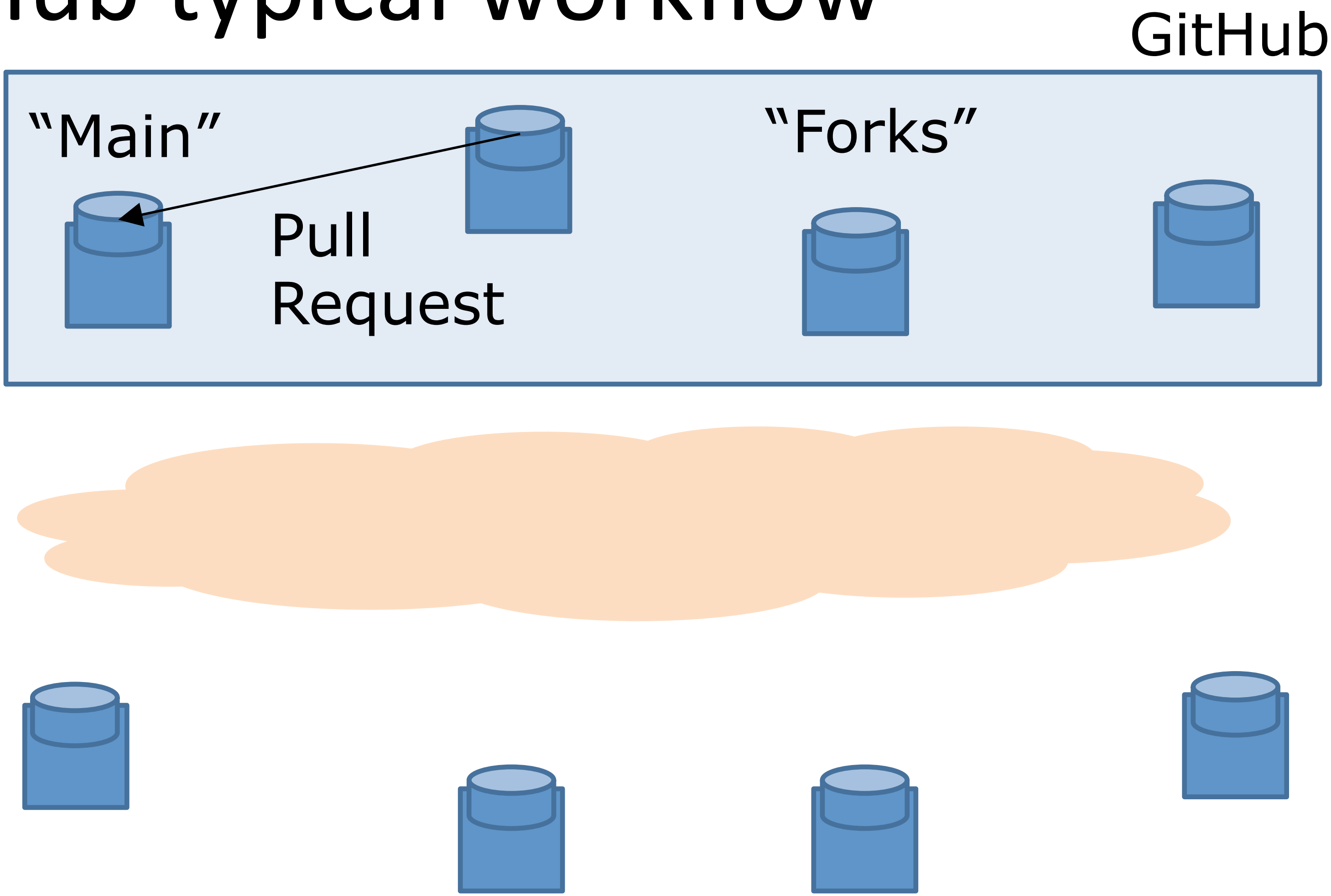
Without access rights, "don't call us, we'll call you" (*pull* from trusted sources) … But again requires host names / IP addresses.

GitHub typical workflow



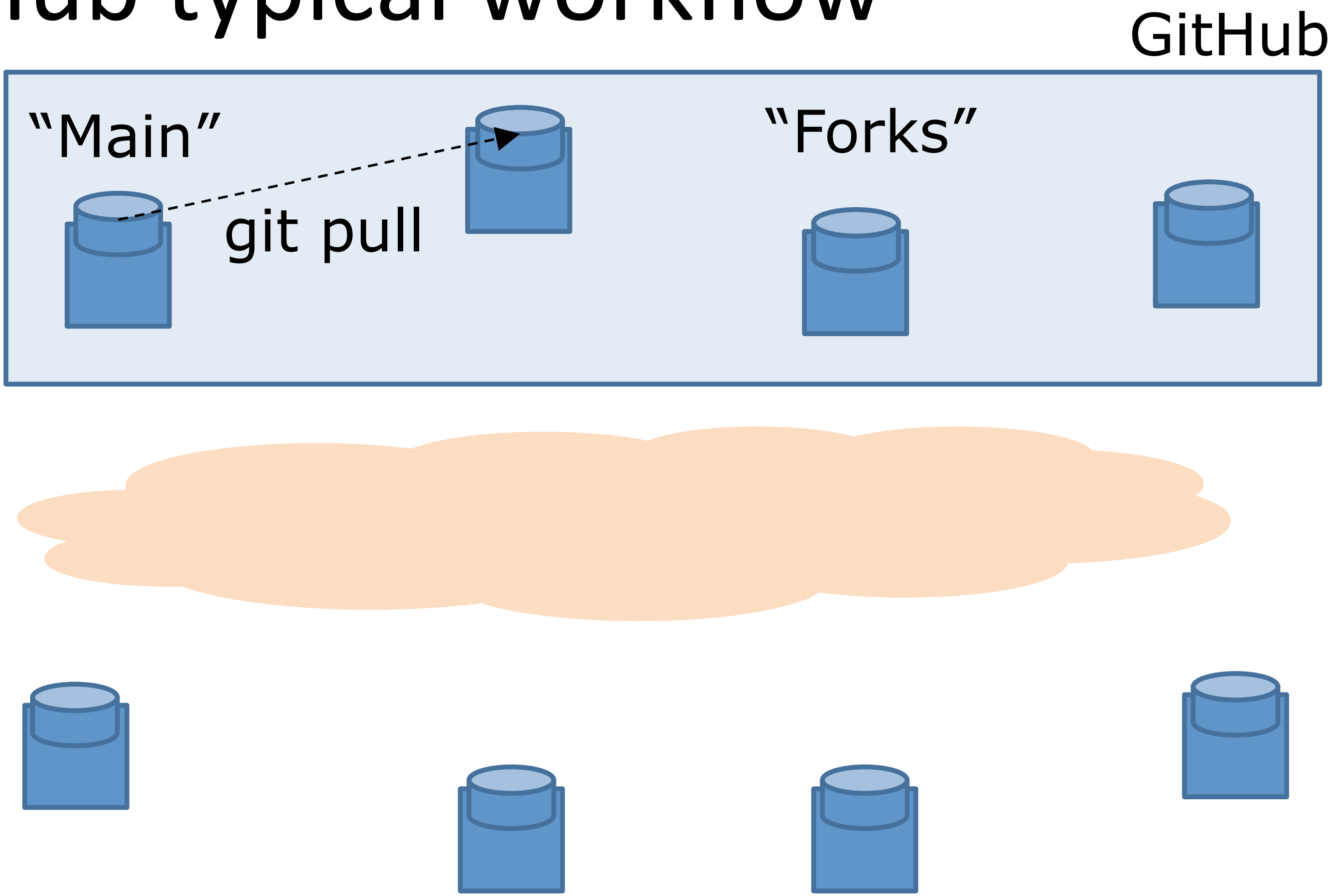
Instead, people maintain public remote "forks" of "main" repository on GitHub and push local changes.

GitHub typical workflow



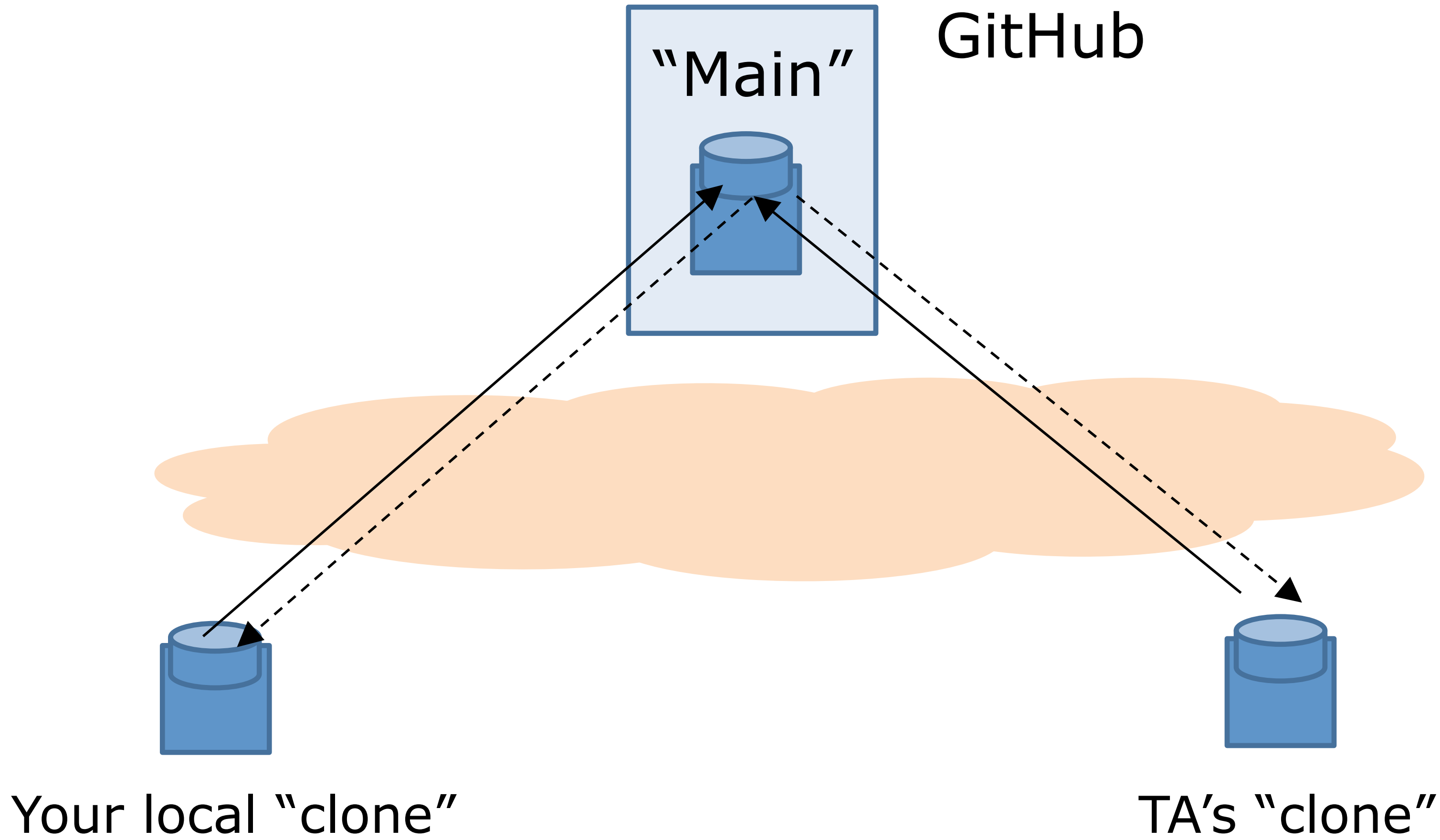
Availability of new changes is signaled via "Pull Request".

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

You will need for homework 1

- Java (+Eclipse/IntelliJ): more on Thursday

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

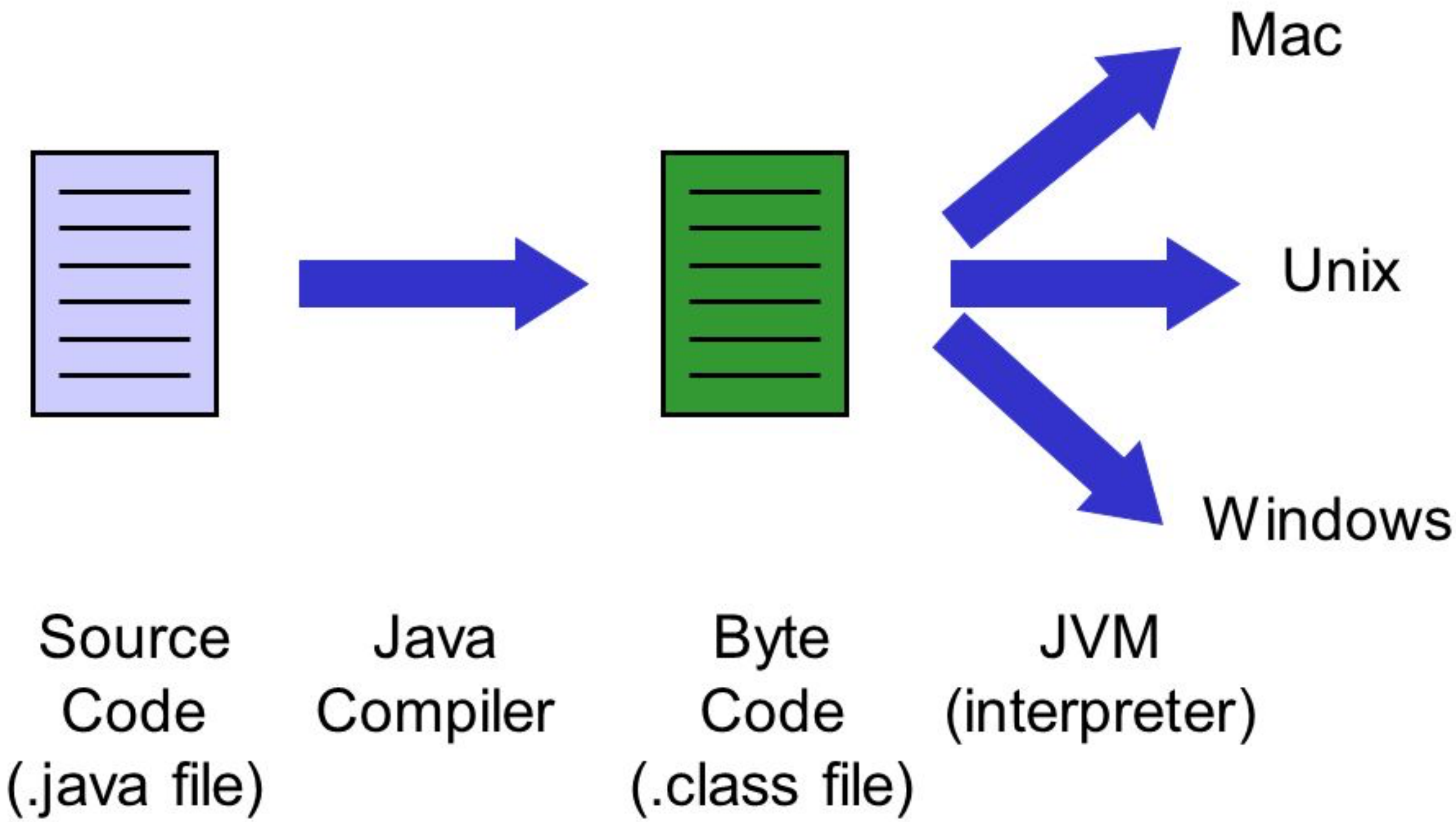
Build Manager

- 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.

Build Manager

- Tool for scripting the automated steps required to produce a software artifact, e.g.:
  - Compile Java source files into class files
  - Compile Java test files
  - Run JUnit tests
  - If all tests pass, package compiled classes into .jar file.

Aside: Java virtual machine



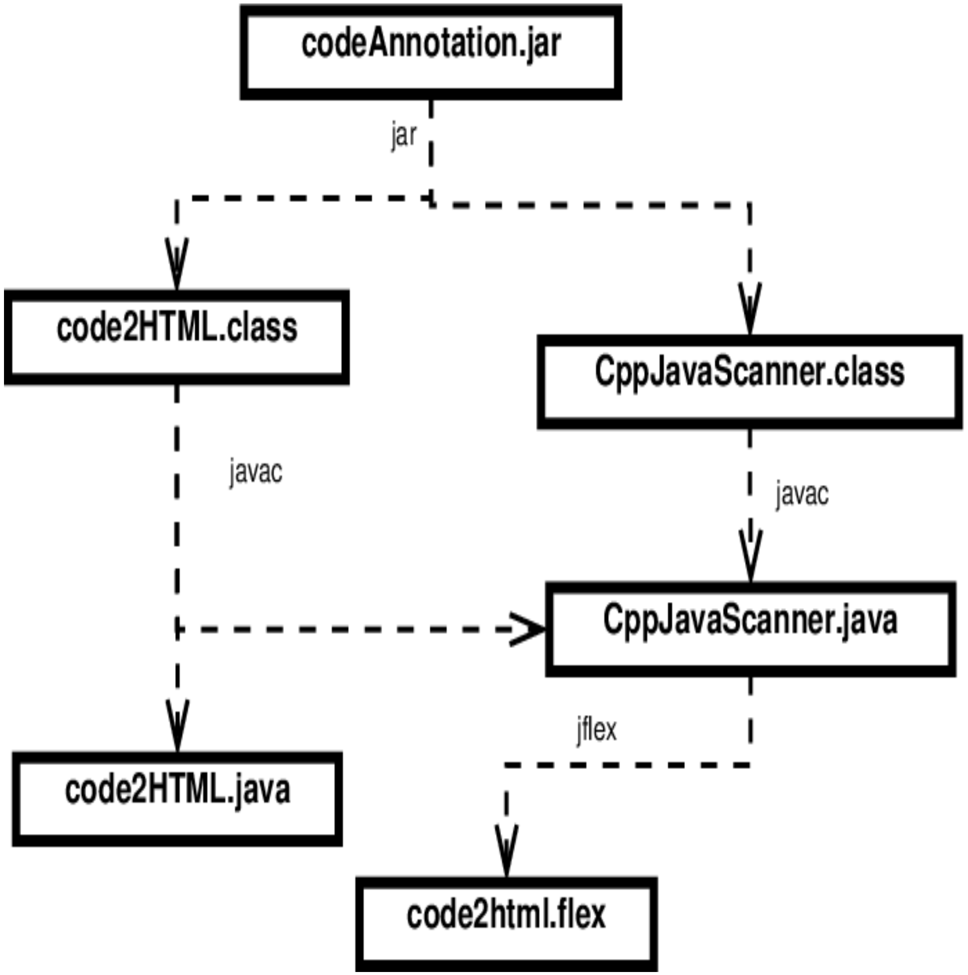
http://images.slideplayer.com/21/6322821/slides/slide_9.jpg

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)

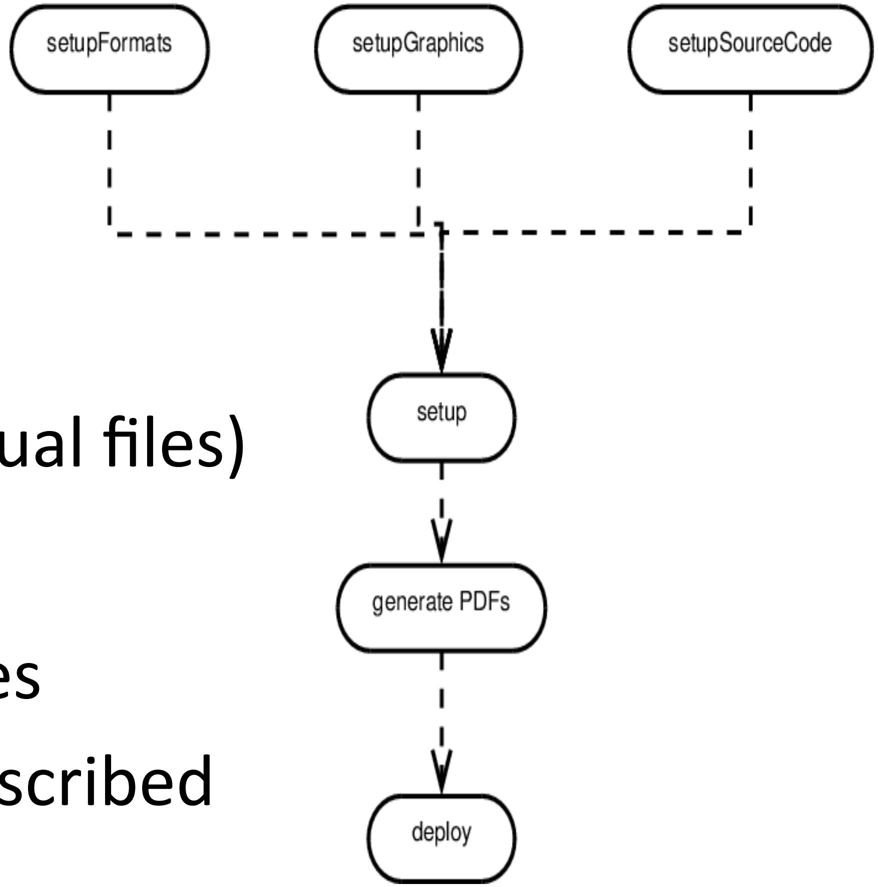
Dependency-Based Managers

- 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.



Task-Based Managers: Ant

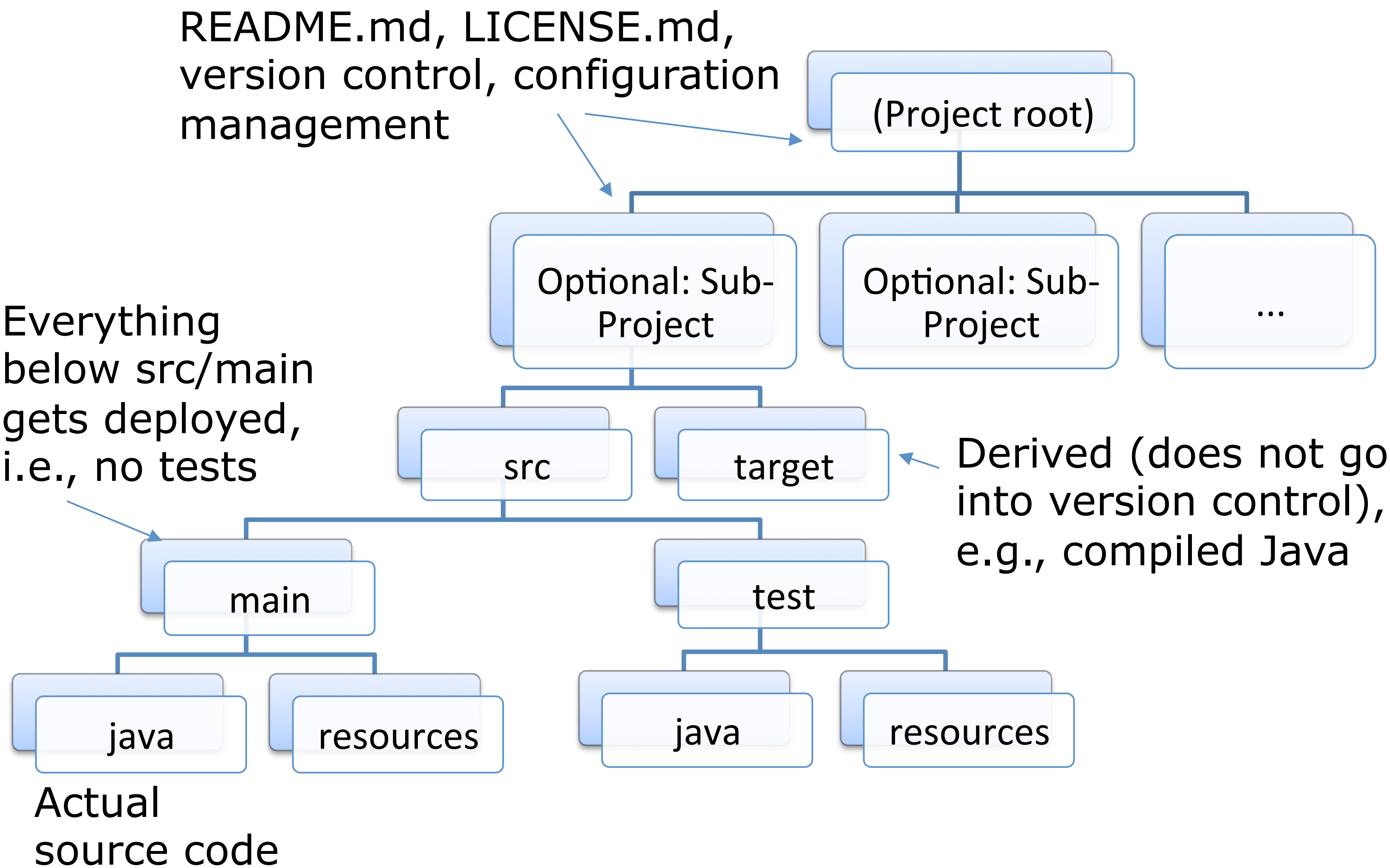
- Disadvantages of Make:
  - Not portable (system-dependent commands, paths, path lists)
  - Low level (focus on individual files)
- Ant:
  - Focus on task dependencies
  - Targets (dependencies) described in build.xml



Task-Based Managers: Maven

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

Organizing a Java Project



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

You will need for homework 1

- Java (+Eclipse/IntelliJ): more on Thursday

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

Big Builds

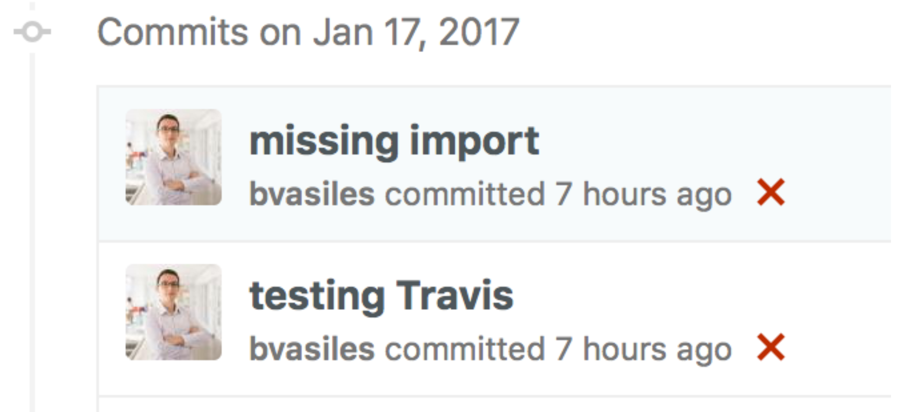
- Must run frequently:
    - fetching and setup of 3rd party libraries
    - static analysis
    - compilation
    - unit testing
    - packaging of artifacts
- Can run less frequently:
    - documentation
    - deployment
    - integration testing
    - test coverage reporting
    - system testing
- 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)

You will need for homework 1

- Java (+Eclipse/IntelliJ): more on Thursday

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