Generating Instructions in Virtual Environments

Session 3: GIVE in practice; tool support

Alexander Koller

29 October 2009

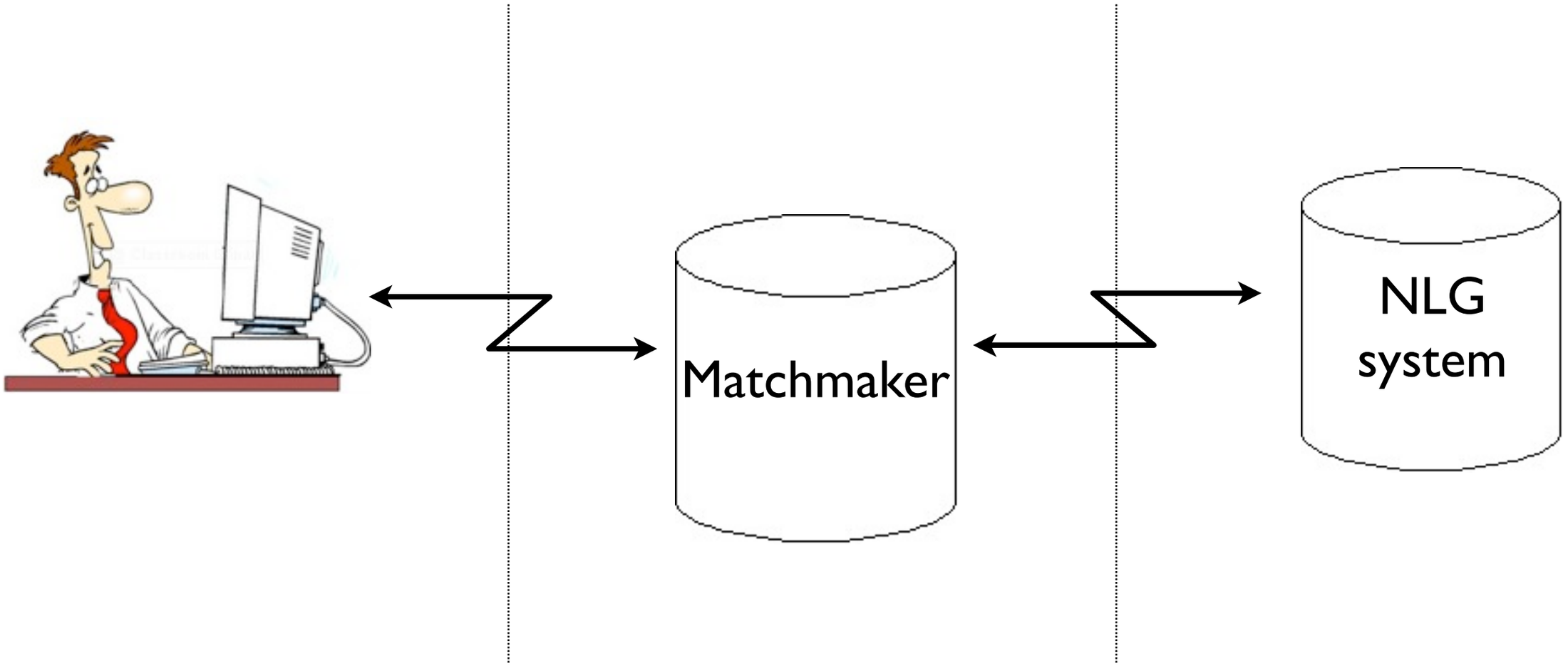
NLG Evaluation

- Evaluating NLG systems is hard.
  - gold standard not meaningful
  - task-based evaluations expensive
- Motivations for GIVE:
  - cheap task-based evaluation
  - focus on situated, real-time communication

Instruction giving in virtual worlds

- Task: Generate real-time instructions that help user perform some task in a virtual environment.
- Use for end-to-end evaluation of NLG systems.

Evaluation



- User and NLG system can be in different places.
- Can perform “web experiments”!

Related Applications

Pedestrian navigation

Task instructions ("Apollo 13")

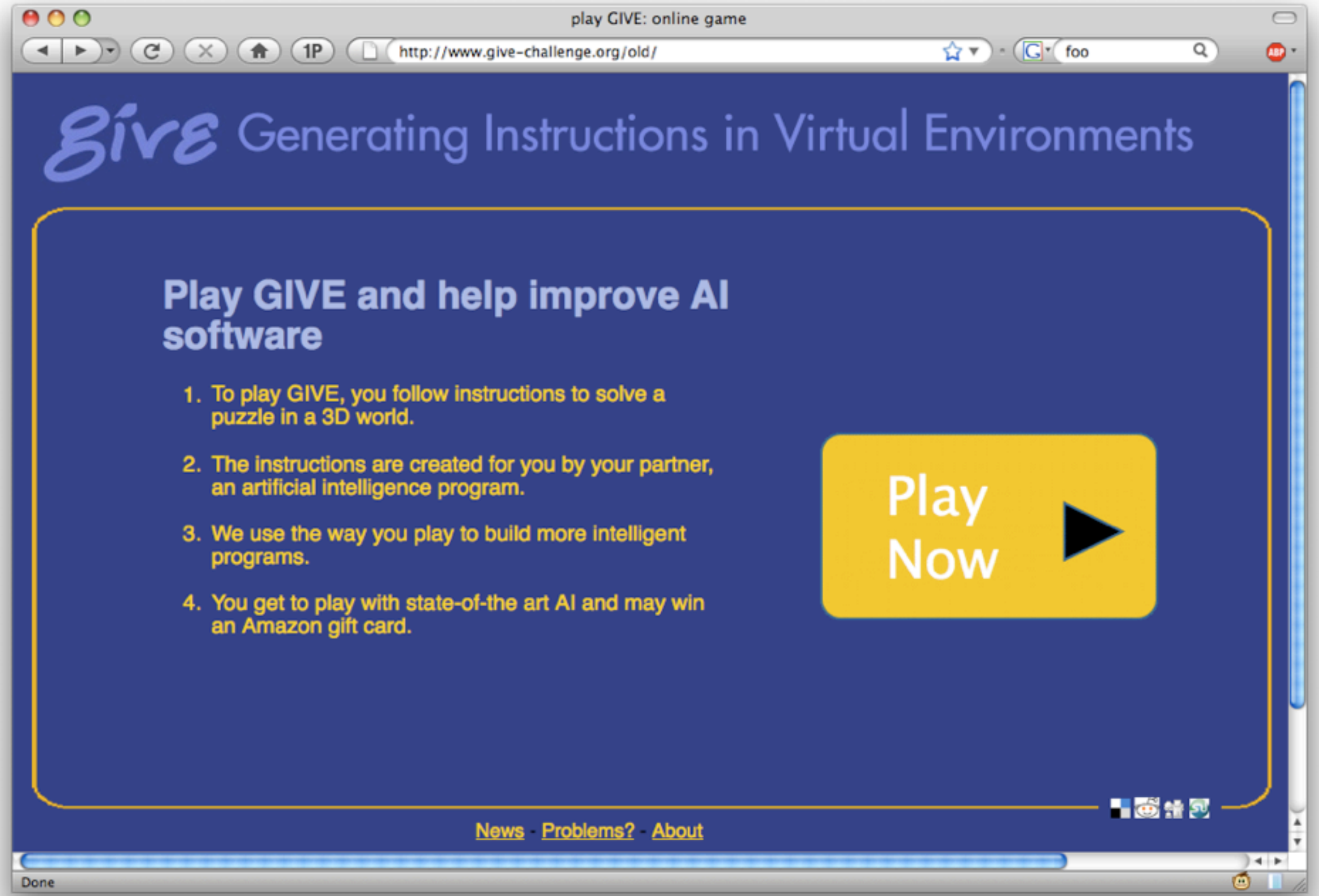
"In vitro" human-robot interaction

GIVE-1

- For the first installment of the challenge:
  - pilot experiment character
  - discrete virtual worlds

- Timeline:
  - announced in March 2008
  - distributed software to participants in May 2008
  - Internet-based evaluation Nov 2008 to Feb 2009
  - data analysis and report writing until March 2009
  - results presented at ENLG in Athens, March 2009

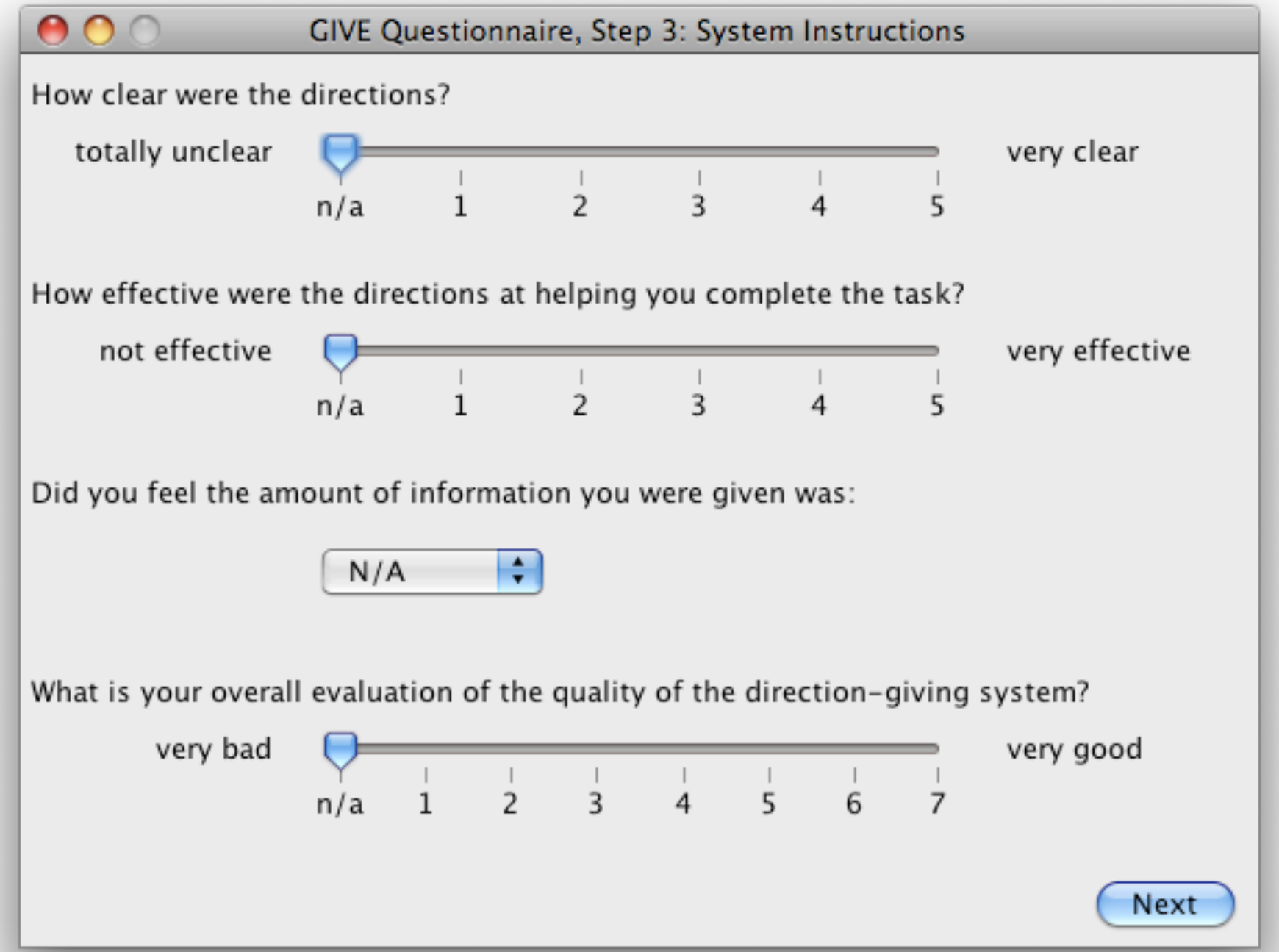
GIVE website



Game client


GIVE 3D Client
Move forward 2 steps, and then press the blue button.
Walk
Turn
Esc
quit
H
What?

Questionnaire



Participating Systems

- Proof-of-concept system: Compute domain plan, realize plan actions one by one.
- Austin: Optimized version of this system (improved paths; some aggregation).
- Madrid: Emphasis on inferring and exploiting "hidden" aspects of world, such as rooms, corners, etc.

Participating Systems (2)

- Union College: Emphasis on navigation instructions, switches between landmark-based and path-based modes.

- Twente: Emphasis on adaptation to user's ability to understand instructions.

- Twente Warm/Cold system: only says "warmer", "colder", etc.; intended to maximize entertainment.

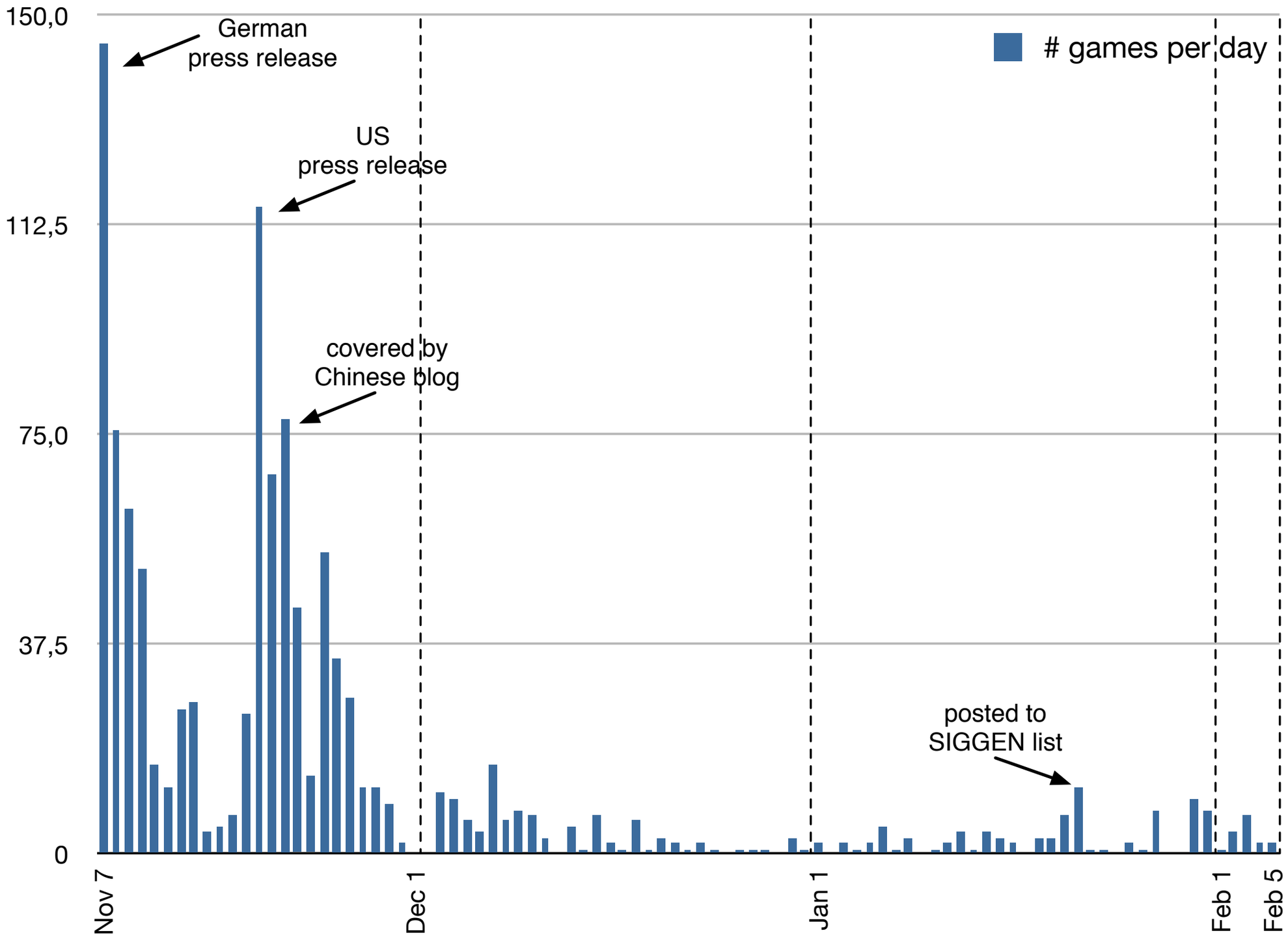
Results

(Connections in 24-hour window on 20 Nov 08)

Results


Greenland
Iceland
Canada
México
Venezuela
Colombia
Perú
Brasil
Brazil
Bolivia
Chile
Argentina
Suomi
Finland
Norge
Norway
Россия
Russia
Kazakhstan
Mongolia
Tanzania
Angola
Namibia
Botswana
Madagascar
South Africa
Indian Ocean
Australia
Papua New Guinea
New Zealand
North Pacific Ocean
South Pacific Ocean
South Atlantic Ocean
Collected 1143 valid games over 3-month period
(Connections in 24-hour window on 20 Nov 08)
Map data ©2008 Europa Technologies - Terms of Use

Results: Timeline



Results: Objective measures

	A	M	T	U	W
task success	40% B	71% A	35% B	73% A	18% C
instructions	83.2 B	58.3 A	121.2 C	80.3 B	190.0 D
steps	103.6 A	124.3 B	160.9 C	117.5 A B	307.4 D
actions	11.2 B	8.7 A	14.3 C	9.0 A	14.3 C
seconds	129.3 A	174.8 B	207.0 C	175.2 B	312.2 D

Differences are significant if two systems don't share a letter.

Lower letters are better.

Results: Subjective measures

	A	M	T	U	W
overall	4.9 A	4.9 A	4.3 B	4.6 A B	3.6 C
choice of words	4.2 A	3.8 B C	4.1 A B	3.7 C	3.5 C
referring expressions	3.4 B	3.9 A 	3.7 A B	3.7 A B	3.5 B
navigation instructions	4.6 A	4.0 B	4.0 B	3.7 B	3.2 C
timing	78% A	62% B	60% B C	62% B	49% C
friendliness	3.4 A B	3.8 A 	3.1 B	3.6 A 	3.1 B

	A	M	T	U	W
task difficulty	4.3 A	4.3 A	4.0 A	4.3 A	3.5 B
goal clarity	4.0 A	3.7 A	3.9 A	3.7 A	3.3 B
play again	2.8 A	2.6 A	2.4 A	2.9 A	2.5 A
instruction clarity	4.0 A	3.6 A B	3.8 A B	3.6 B	3.0 C
instruction helpfulness	3.8 A	3.9 A	3.6 A	3.7 A	2.9 B
informativity	46% B	68% A 	51% B	56% B	51% B

overall on 1-7 scale;
timing, informativity "just right" vs. not;
all others on 1-5 scale.

Summary: GIVE

- GIVE-1 was largest evaluation effort for NLG systems in terms of users, ever.
- Evaluated 5 systems, which emphasized different aspects. Significant differences, consistent with lab experiment.
- Simple systems work surprisingly well.

GIVE-2

- Mostly like GIVE-1, but:
  - continuous worlds
  - improved evaluation measures

- Development phase started in August.

- Evaluation phase is Feb - Apr 10.

- Presentation of results in July 10 at INLG.

Structure of GIVE software

3D Client

NLG Server

Matchmaker

Structure of GIVE software

3D Client

connection

Matchmaker

NLG Server

Structure of GIVE software

3D Client

connection

Matchmaker

connection

NLG Server

Structure of GIVE software

3D Client

instructions

updates about events

connection

connection

NLG Server

Matchmaker

Structure of GIVE software

3D Client

instructions

updates about events

connection

Matchmaker

connection

NLG Server

game log

Structure of GIVE software

3D Client

instructions

updates about events

connection

NLG Server

connection

game log

Matchmaker

Database

Structure of GIVE software

3D Client

instructions

updates about events

connection

questionnaire

NLG Server

connection

game log

Matchmaker

Database

Structure of GIVE software

3D Client

instructions

updates about events

connection

questionnaire

web interface

etc.

connection

game log

Matchmaker

NLG Server

Database

Structure of GIVE software

3D Client

instructions

updates about events

connection

questionnaire

web interface

etc.

connection

game log

Matchmaker

NLG Server

NLG System

Database

Structure of GIVE software

3D Client

instructions

updates about events

questionnaire

connection

web interface

etc.

connection

game log

Matchmaker

NLG Server

NLG System

you are here

Database

Writing a GIVE NLG system

- Derive a Java class from the abstract class give.nlgserver.NlgSystem in the GIVE API.
- Write a configuration file that tells the GIVE NLG server where to find your NLG system.
- Start the GIVE NLG server and point your Matchmaker to it.

The class NlgSystem

```
abstract public class NlgSystem {
  abstract public void connectionEstablished(QuestionnaireData preQuestionnaire)
                                throws NlgServerException;

  abstract public void connectionDisconnected();

  abstract public void handleStatusInformation(Position playerPosition,
      Orientation playerOrientation, List<String> visibleObjects)
      throws NlgServerException;

  abstract public void handleAction(Atom actionInstance, List<Formula> updates)
                                throws NlgServerException;

  abstract public void handleDidNotUnderstand()
                                throws NlgServerException;
}
```

That's it!

Methods you can use

- void send(String s): Send the string s to the client to be displayed.

- List<Atom> getPlan(List<Formula> goals): Get a plan that leads from current state to goals.

- getWorld(), getDiscretizer(): Get current world and discretizer.

- see: http://give-challenge.kenai.com/apidocs/

Configuring the NLG server

```
<?xml version="1.0"?>

<nlgserver port="3001" simultaneous-instances="1" web-port="8081">
  <nlg-system
       class="edu.union.give2.simplenlgservernolandmarks.ExampleNlgSystem" />

  <planner style="sgplan" executable="/home/koller/sgplan-522-mac" />
</nlgserver>
```

(example-nlgsystem-config.xml in Demo Servers distribution)

Then start the NLG server:

```
$ java -jar target/give2-example-nlgserver-1.0.2-jar-with-dependencies.jar
    example-nlgsystem-config.xml

GIVE NLG server, version null
Reading configuration from example-nlgsystem-config.xml...
NLG system class: edu.union.give2.simplenlgservernolandmarks.ExampleNlgSystem
[Server 20:20:02.699] Listening on port 3001
```

Compiling a GIVE NLG system

- We use the Maven build tool for GIVE-2.
- Maven's perspective on building stuff:
  - declarative configuration file
  - convention about directory structure
  - download libraries automatically when needed
- The last point makes it very convenient for GIVE.

Maven directory structure

```
myproject/
  pom.xml          -- the central configuration file
  /src
    /main          -- source code that gets compiled into Jar files
      /java        -- Java source code
      /javacc      -- JavaCC source code (parse generator)
      /resources -- any other files that will be included in Jar files
      ...
    /test          -- source code for unit tests
      /java        -- ... written in Java
      /groovy      -- ... written in Groovy
      ...
  /target          -- compiled classes and Jar files (created by Maven)
```

A pom file (important bits)

Identify your project name and version:

```
    <groupId>edu.union</groupId>
    <artifactId>give2-example-nlgserver</artifactId>
    <name>give2-example-nlgserver</name>
    <version>1.0.2</version>
```

Tell Maven to include the NLG server library:

```
<dependency>
    <groupId>de.saar.penguin</groupId>
    <artifactId>give2-nlgserver</artifactId>
    <version>[1.9.1,)</version>
</dependency>
```

(you can include further dependencies if you like)

(see a demo server pom file for the rest)

A pom file (important bits)

Configure build tools:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
    <archive>
      <manifest>
        <mainClass>give.nlgserver.NlgServer</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

(see a demo server pom file for the rest)

Running Maven

- Compile all classes, build a Jar file containing just your classes, install:

```
mvn install
```

- Build executable Jar file with your classes and all libraries:

```
mvn assembly:assembly
```

- Delete all compiled files:

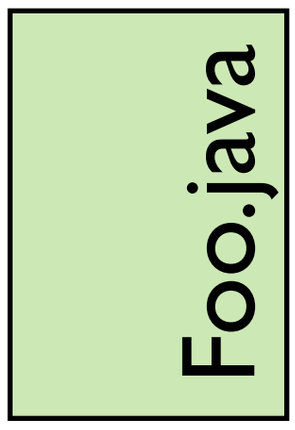
```
mvn clean
```

Collaborating with others

- So far, you have primarily written code by yourself. This project is different.
- Tools for collaborating with other developers:
  - Revision control software: Mercurial
  - Bug tracking, discussions, Wiki: FogBugz
  - Code reviews
  - Project management approach: Scrum

Thank you, Fog Creek Software!

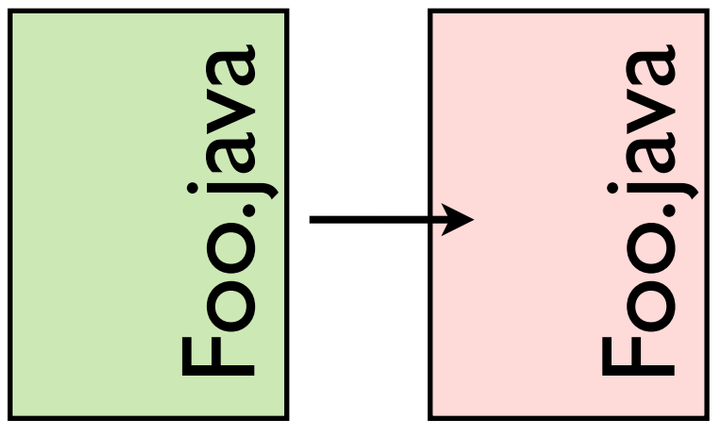
Version control systems



Developer A

Developer B

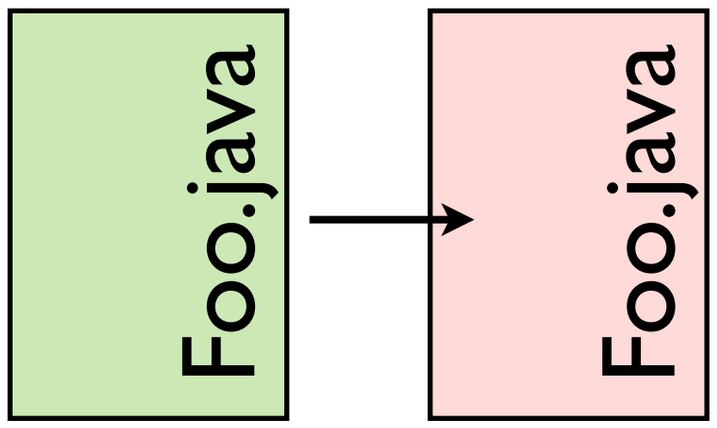
Version control systems



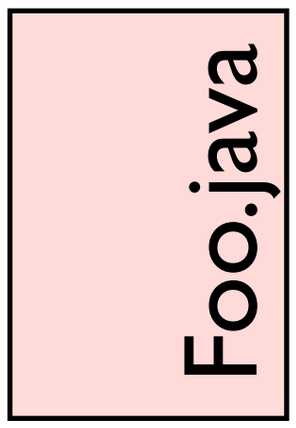
Developer A

Developer B

Version control systems

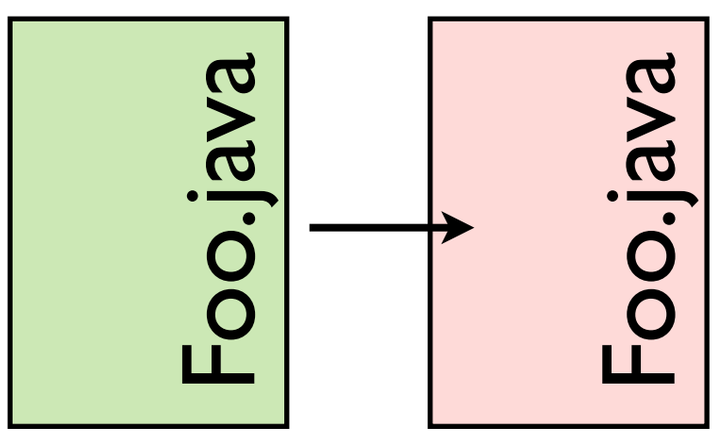


Developer A

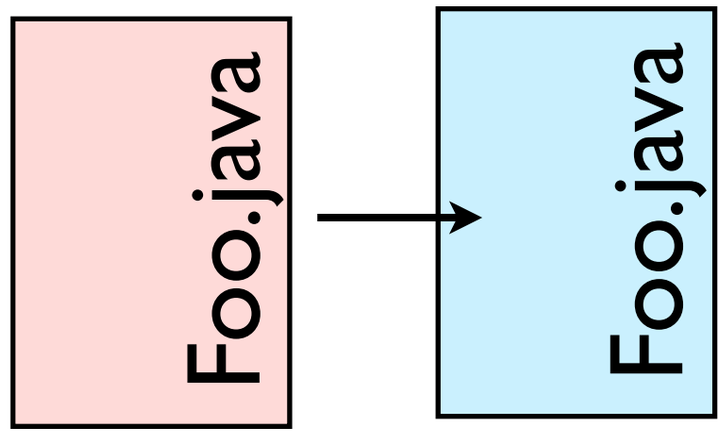


Developer B

Version control systems

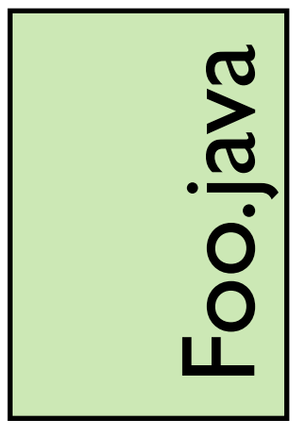


Developer A



Developer B

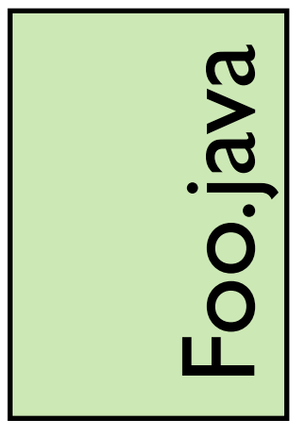
Version control systems



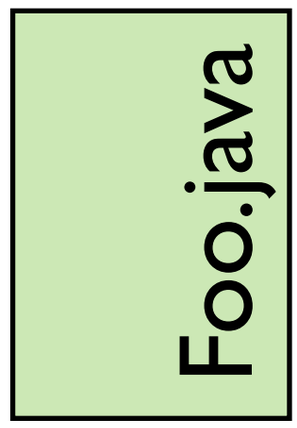
Developer A

Developer B

Version control systems

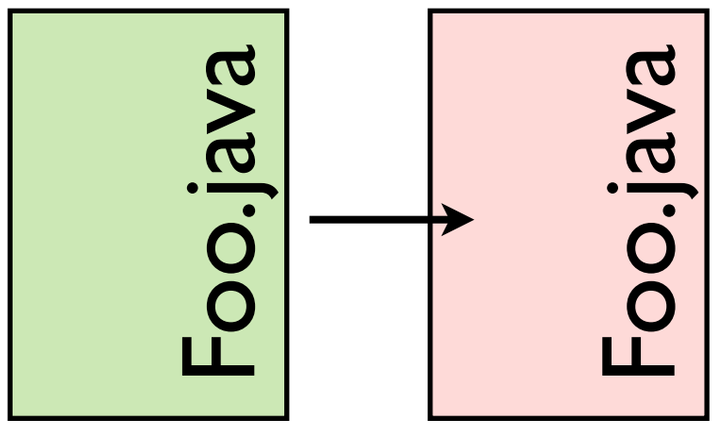


Developer A

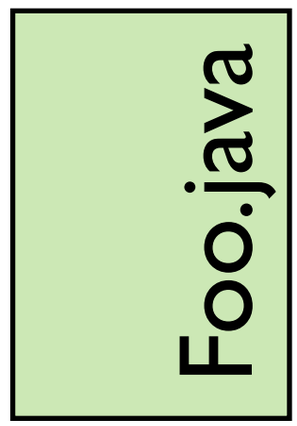


Developer B

Version control systems

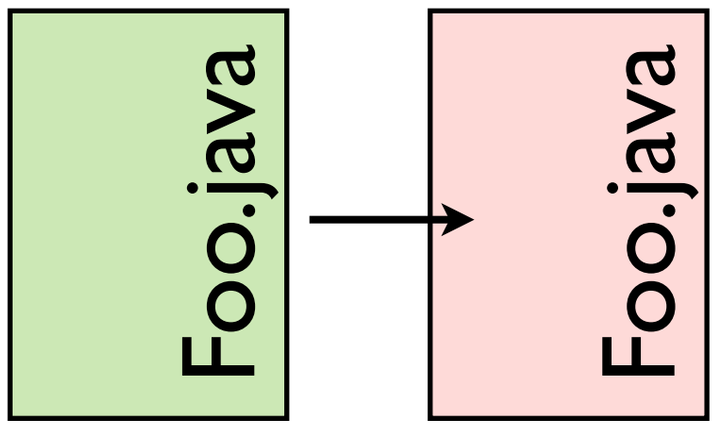


Developer A

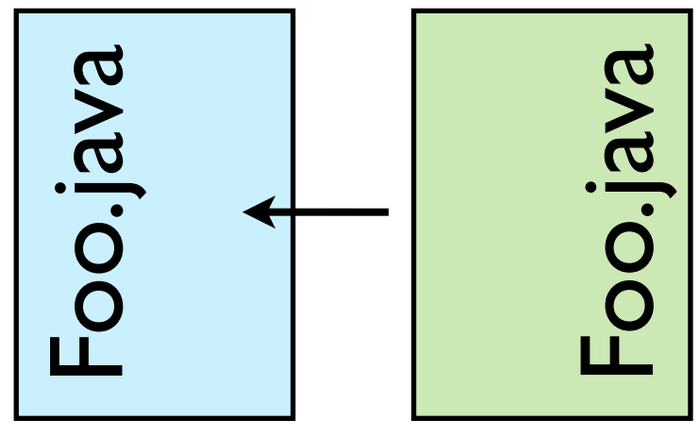


Developer B

Version control systems



Developer A

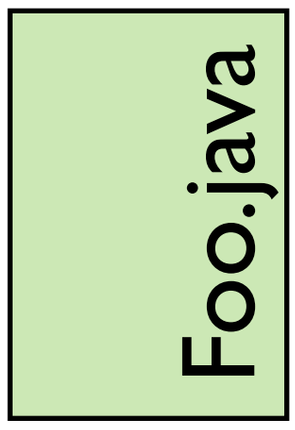


Developer B

Version control systems

Version control systems

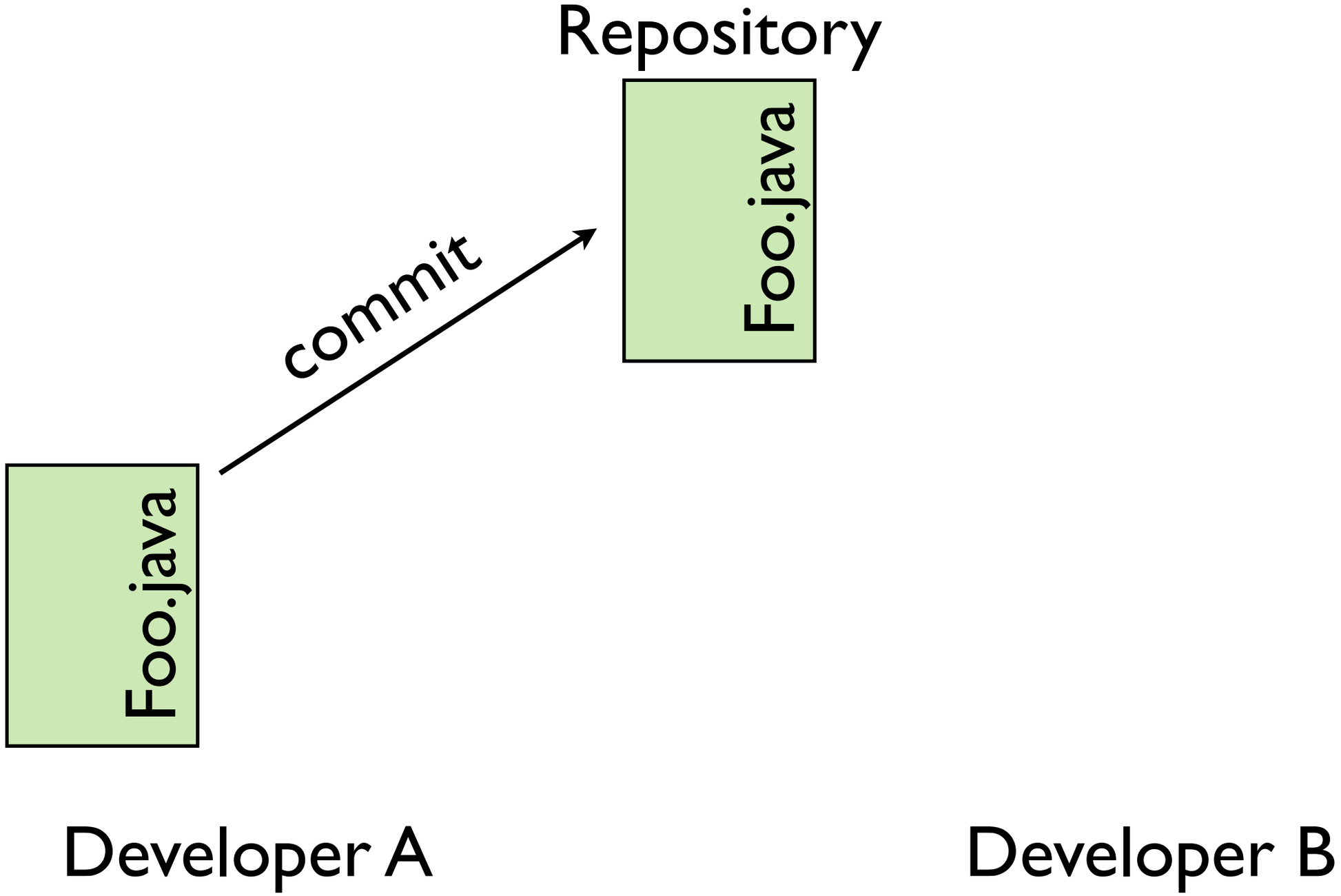
Repository



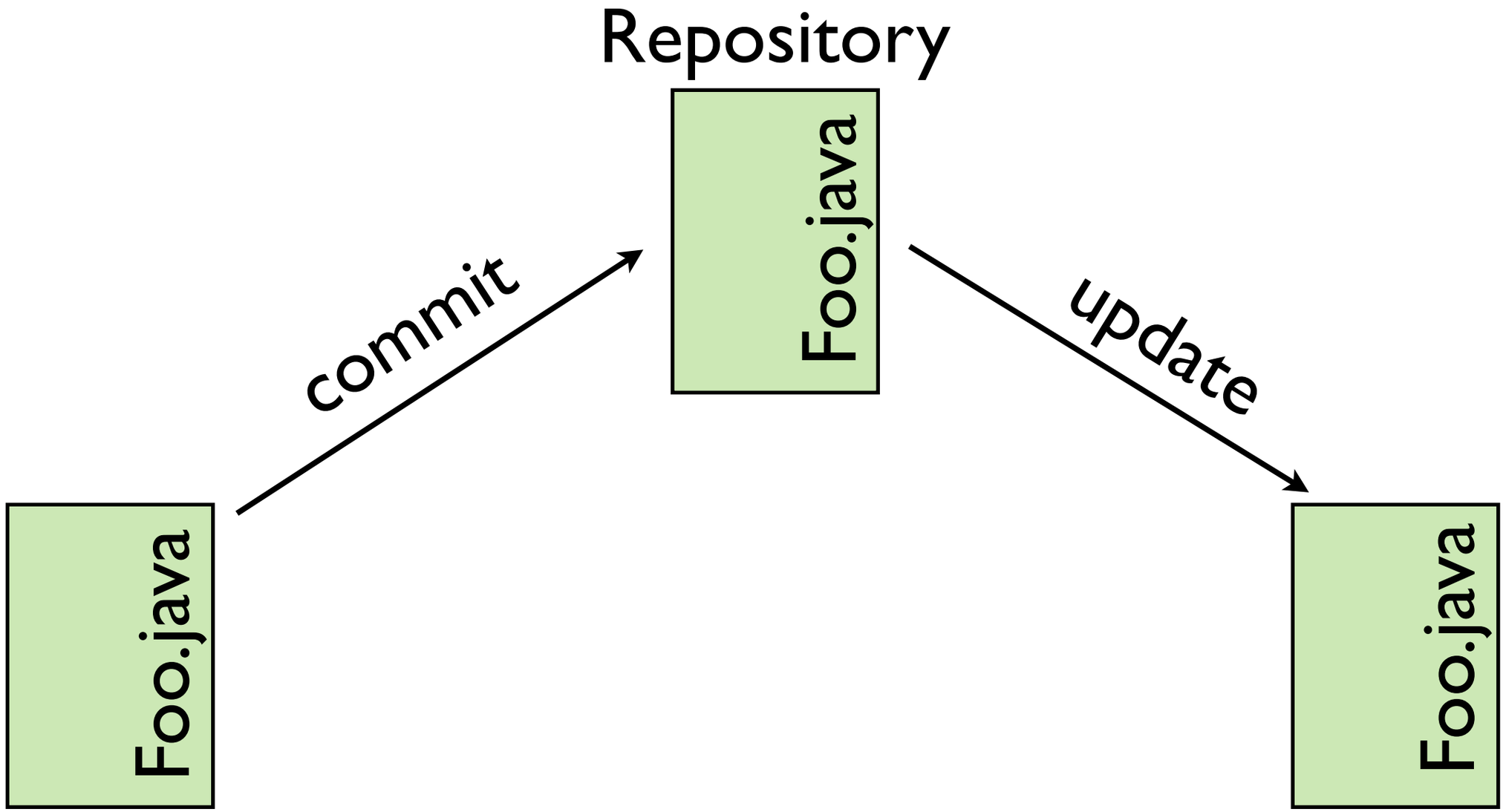
Developer A

Developer B

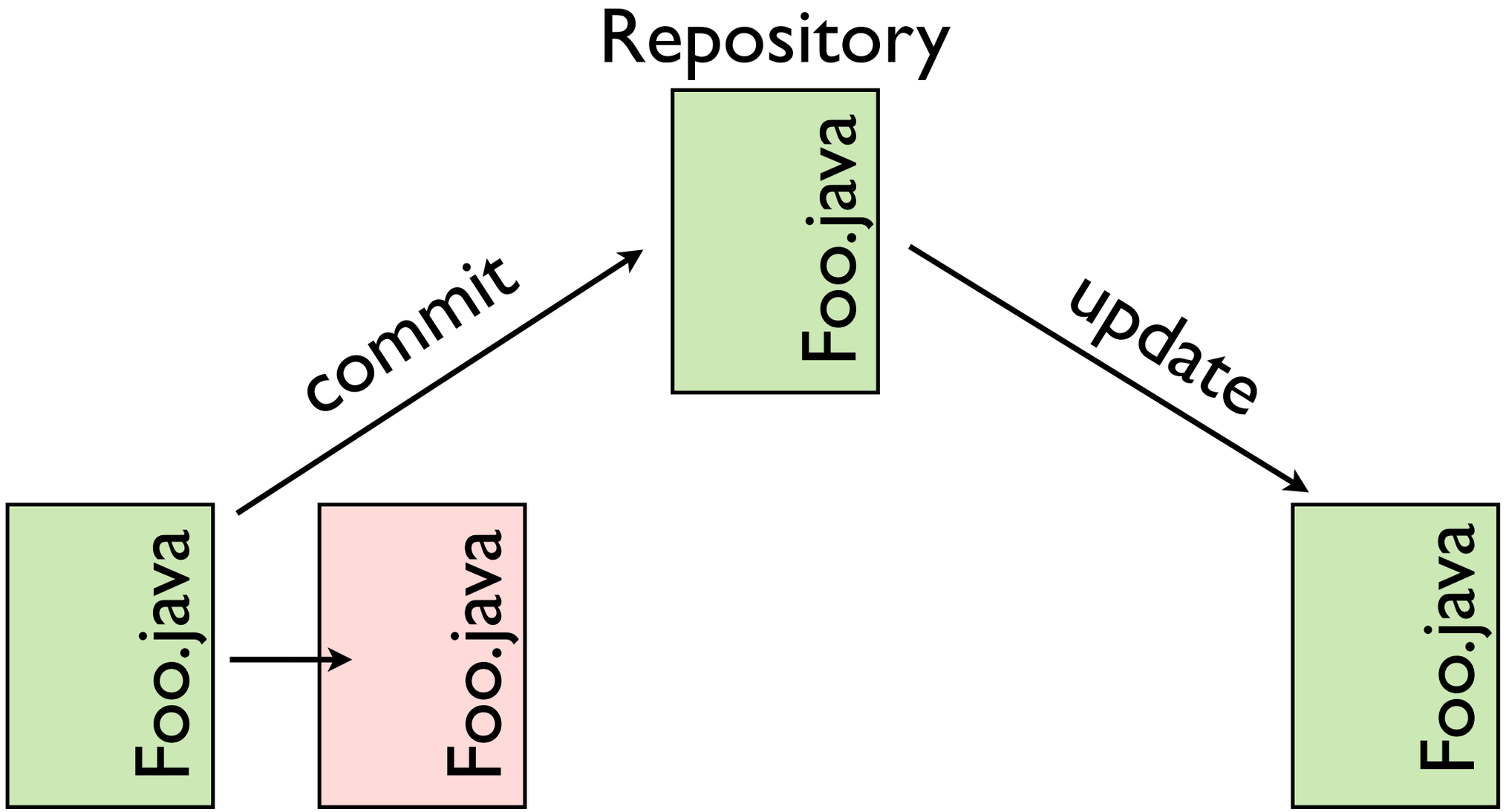
Version control systems



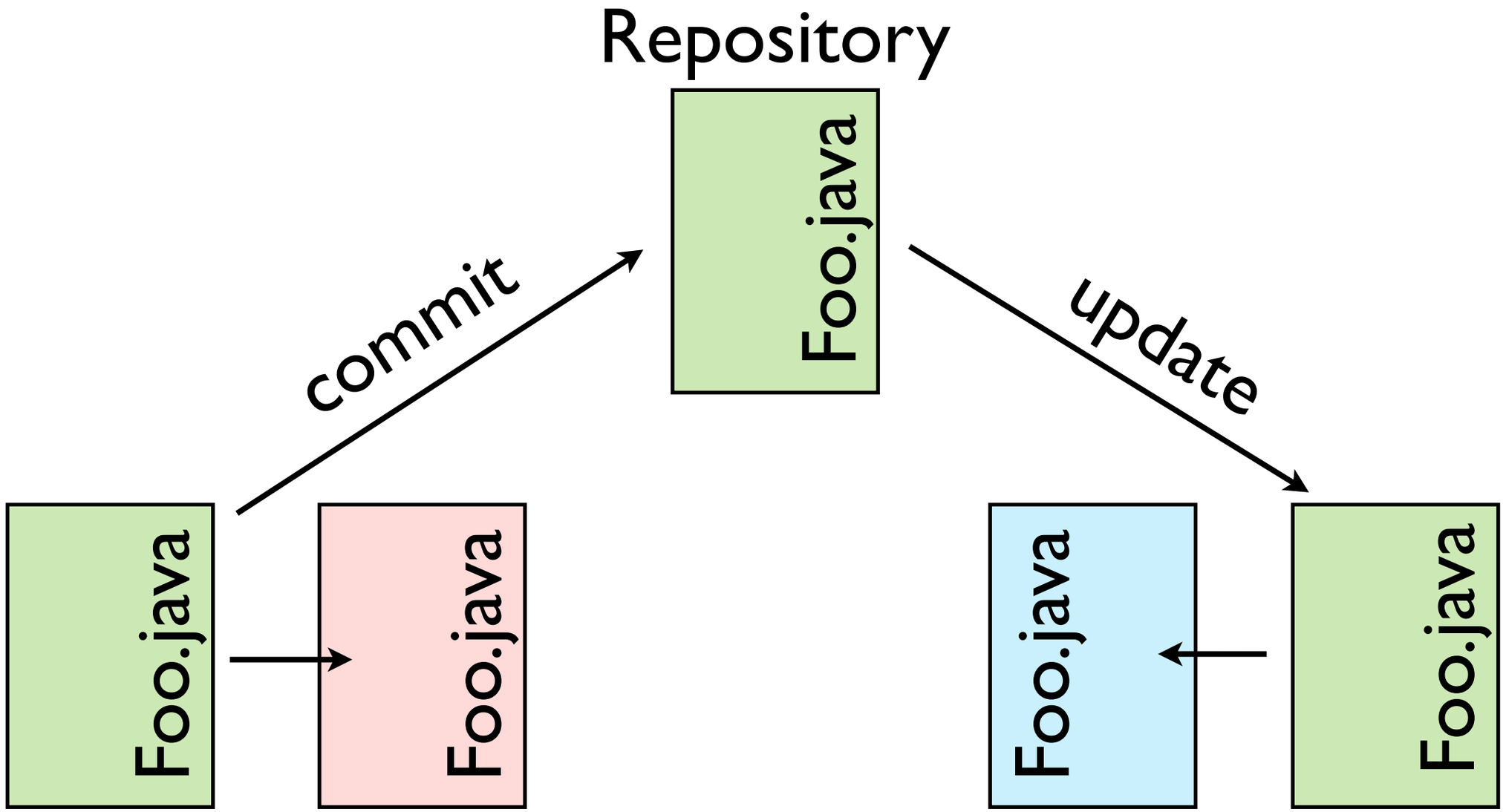
Version control systems



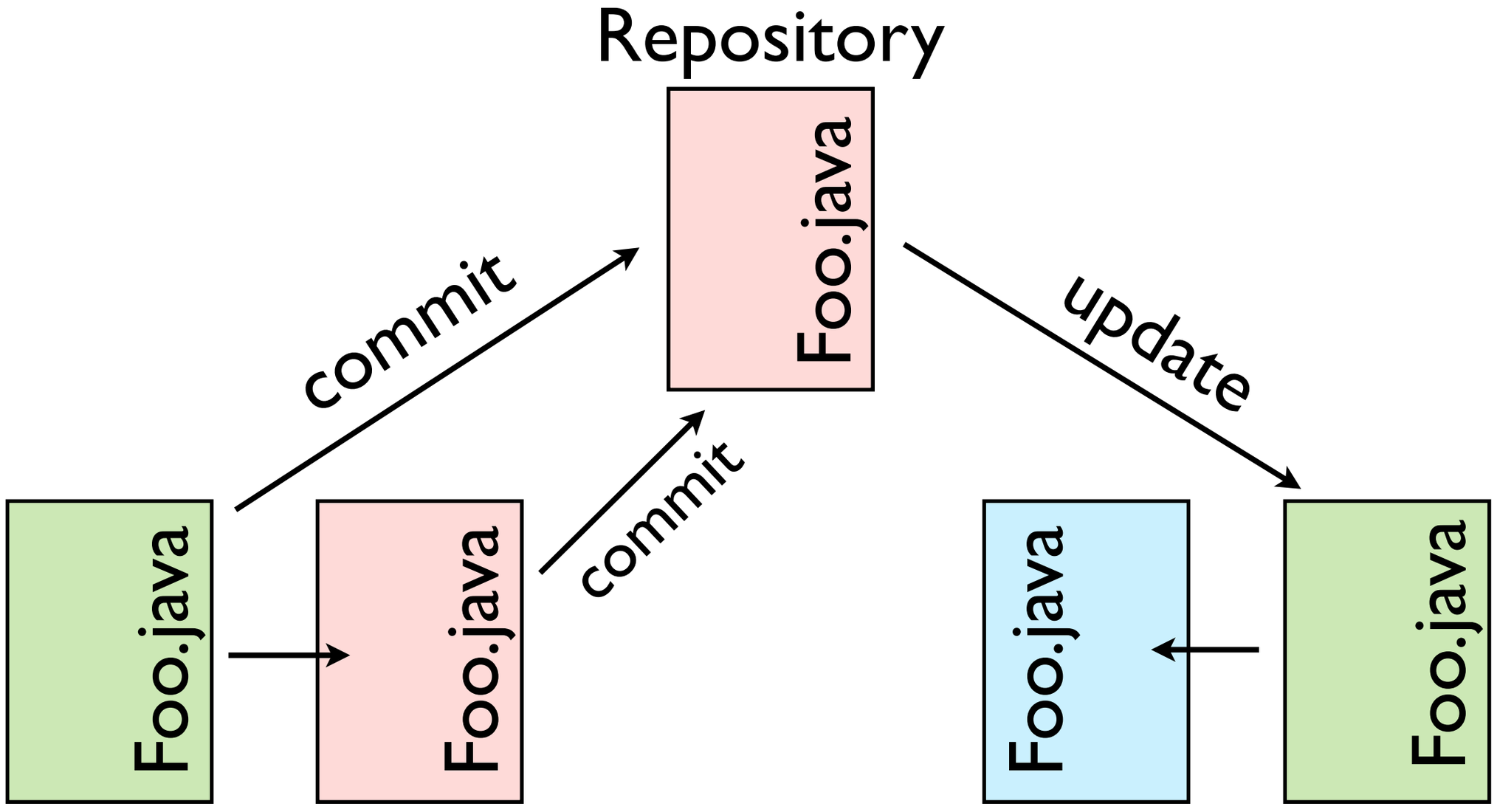
Version control systems



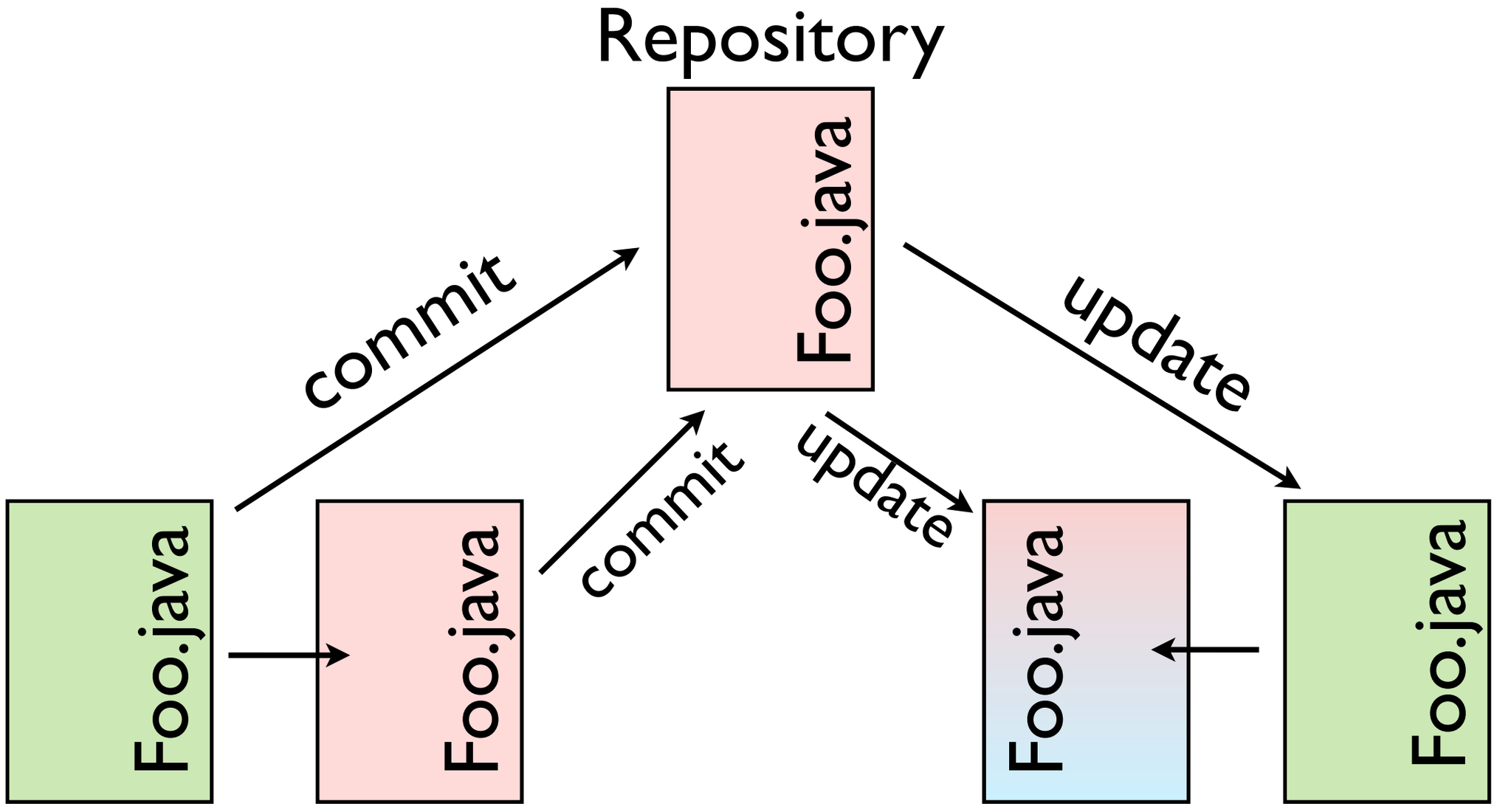
Version control systems



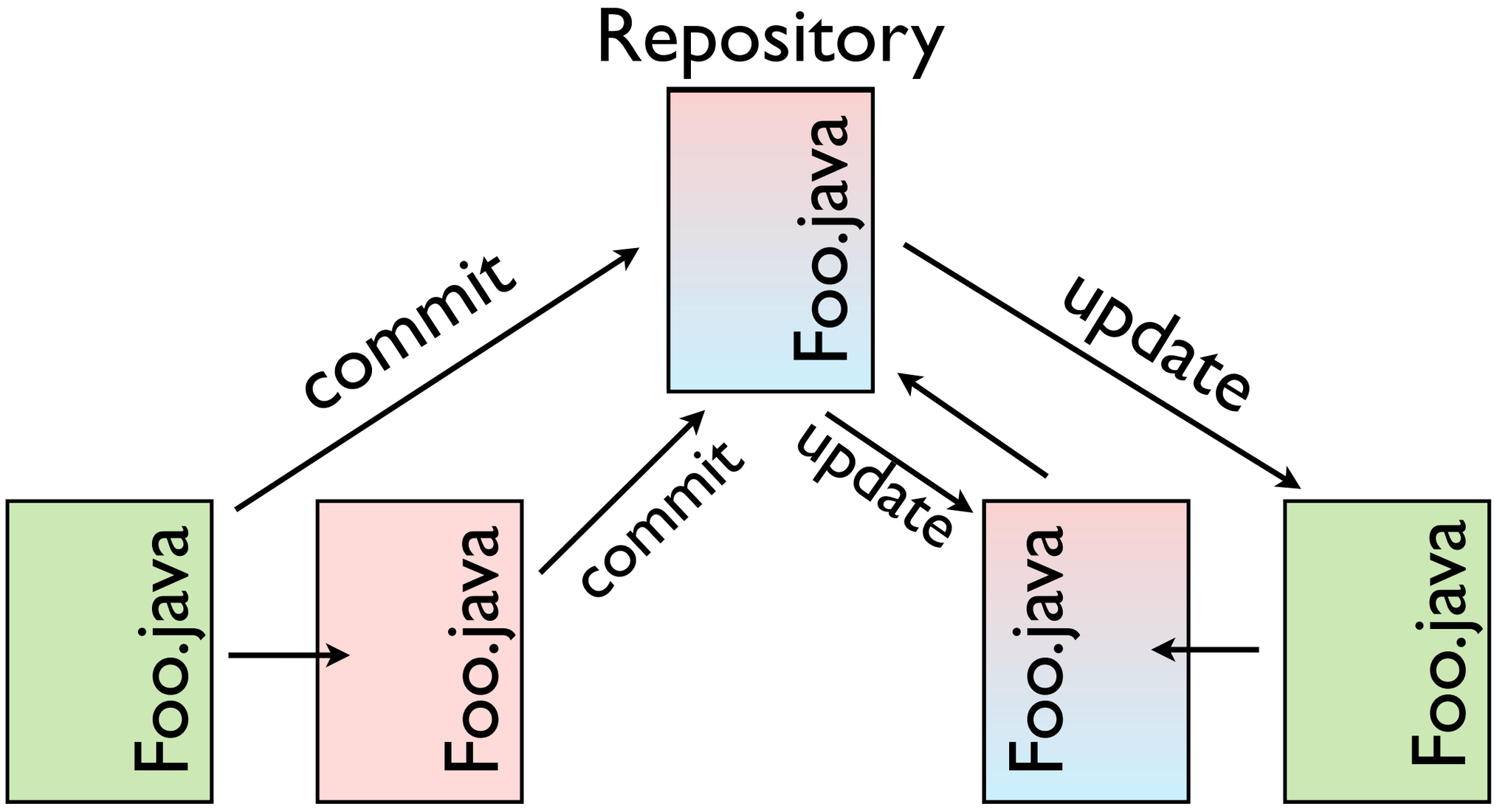
Version control systems



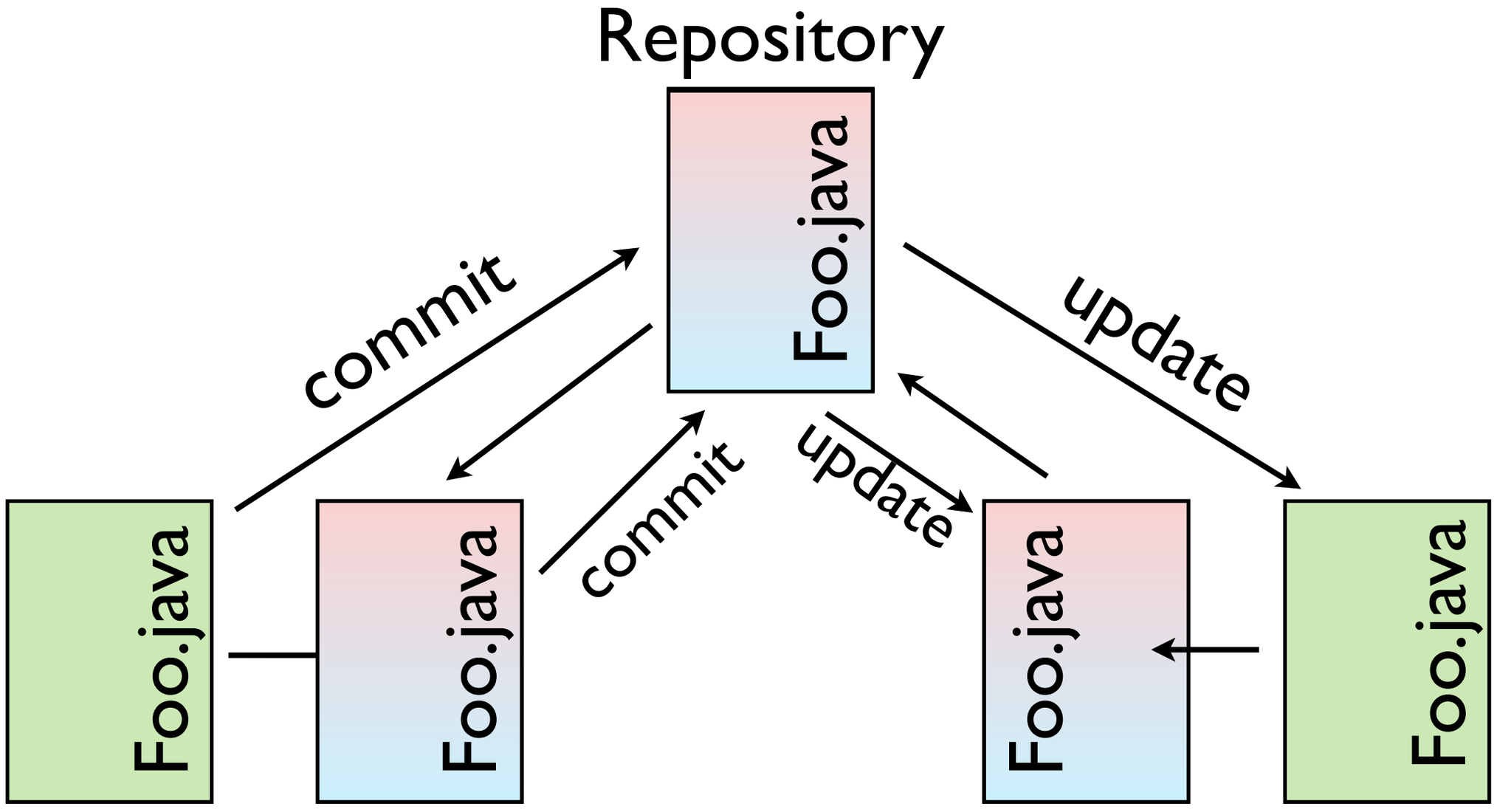
Version control systems



Version control systems



Version control systems



Version control systems

- One central *repository*; each user has their own *working copy*.

- Repository to WC: *update*
  WC to repository: *commit*

- VC system tries to merge changes; if impossible, a *merge conflict* is signalled and has to be resolved manually by user.

We use Mercurial

- ... because that's what our hosting service provides.
- A *decentralized* system: You *can* use the central repository, but you don't *have* to.
- Get info and a client at http://mercurial.selenic.com/
- Most prominent alternative: Subversion.

Using Mercurial

- Clone a repository:

```
hg clone http://hg-scm.org/hello my-hello
```

- After editing your WC, transfer changes back to network repository:

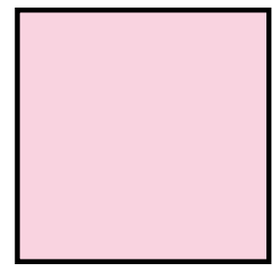
```
hg commit
hg push
```

- Get updates from network repository:

```
hg pull -u
hg merge
```

Mercurial: The model

repositories



working copies

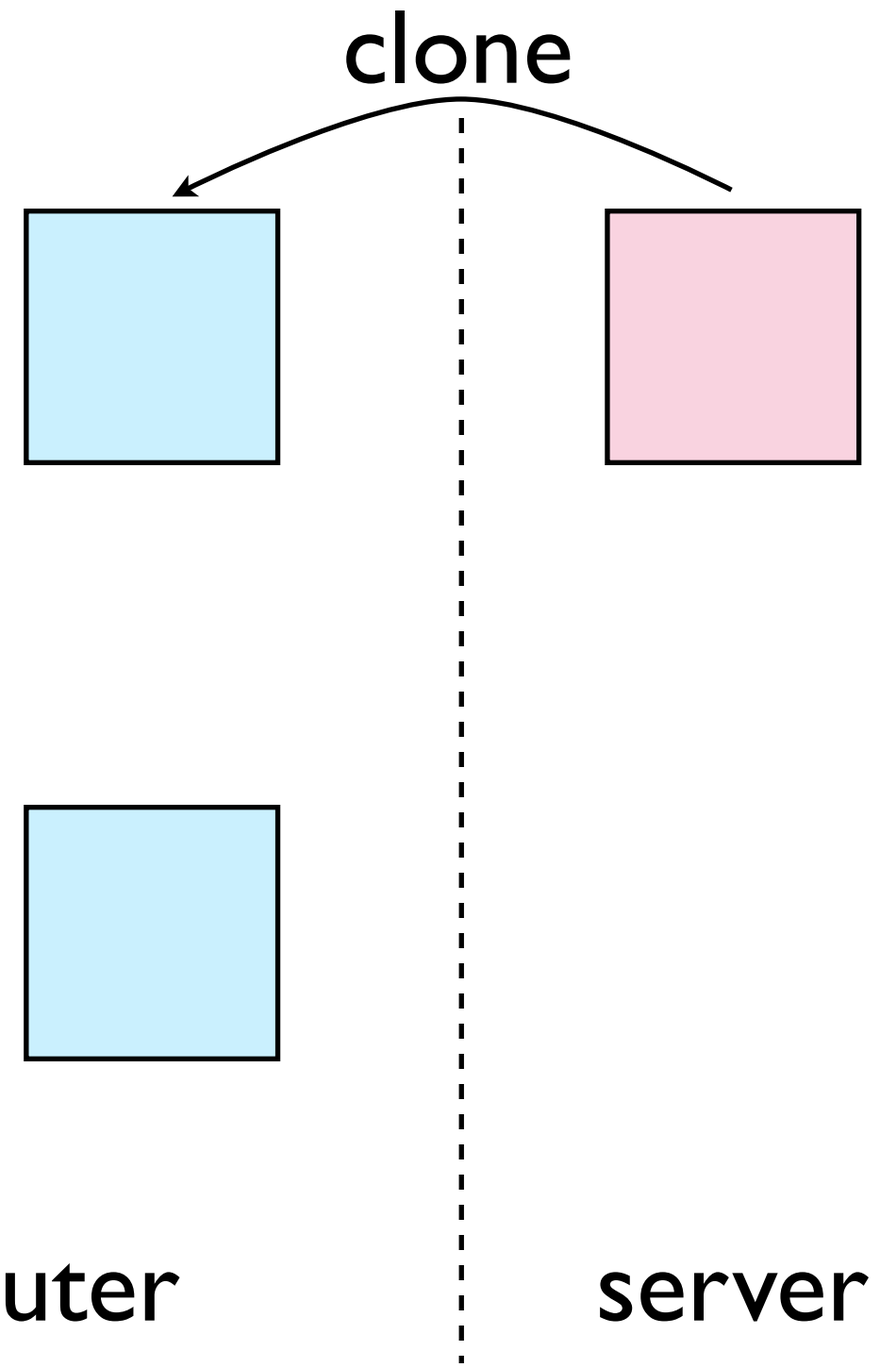
your computer

server

Mercurial: The model

repositories

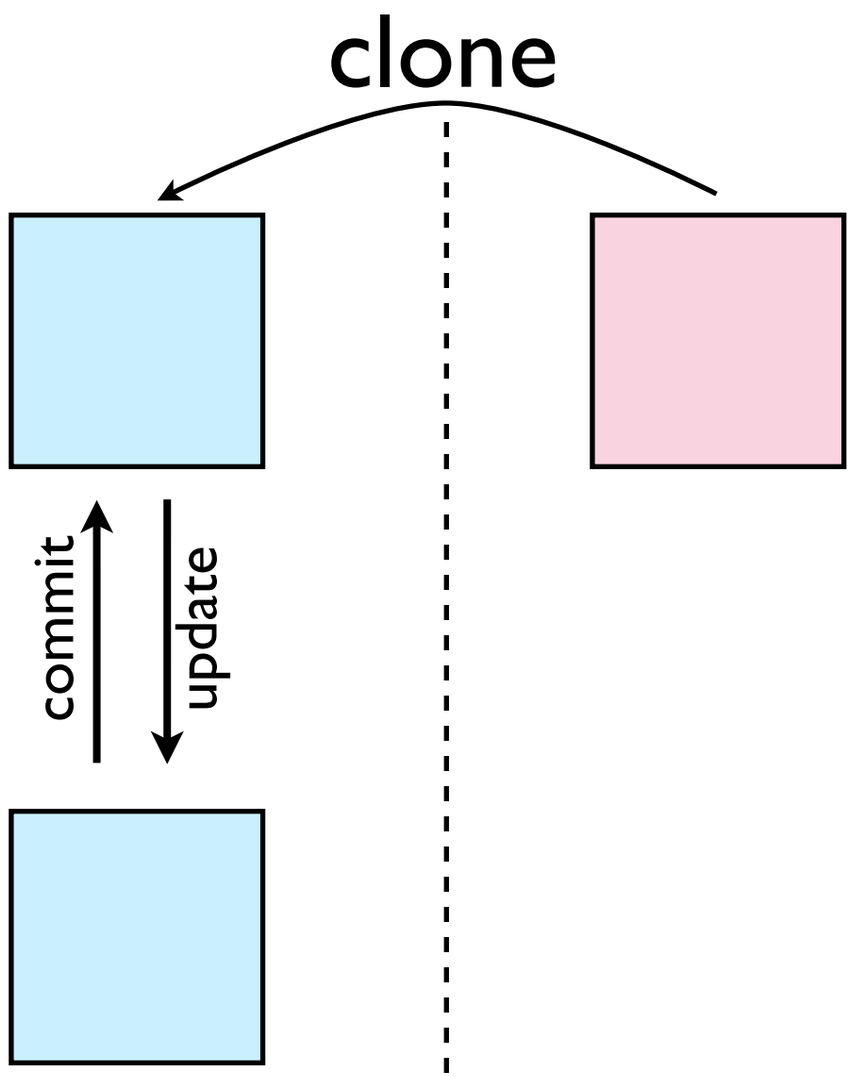
working copies



Mercurial: The model

repositories

working copies



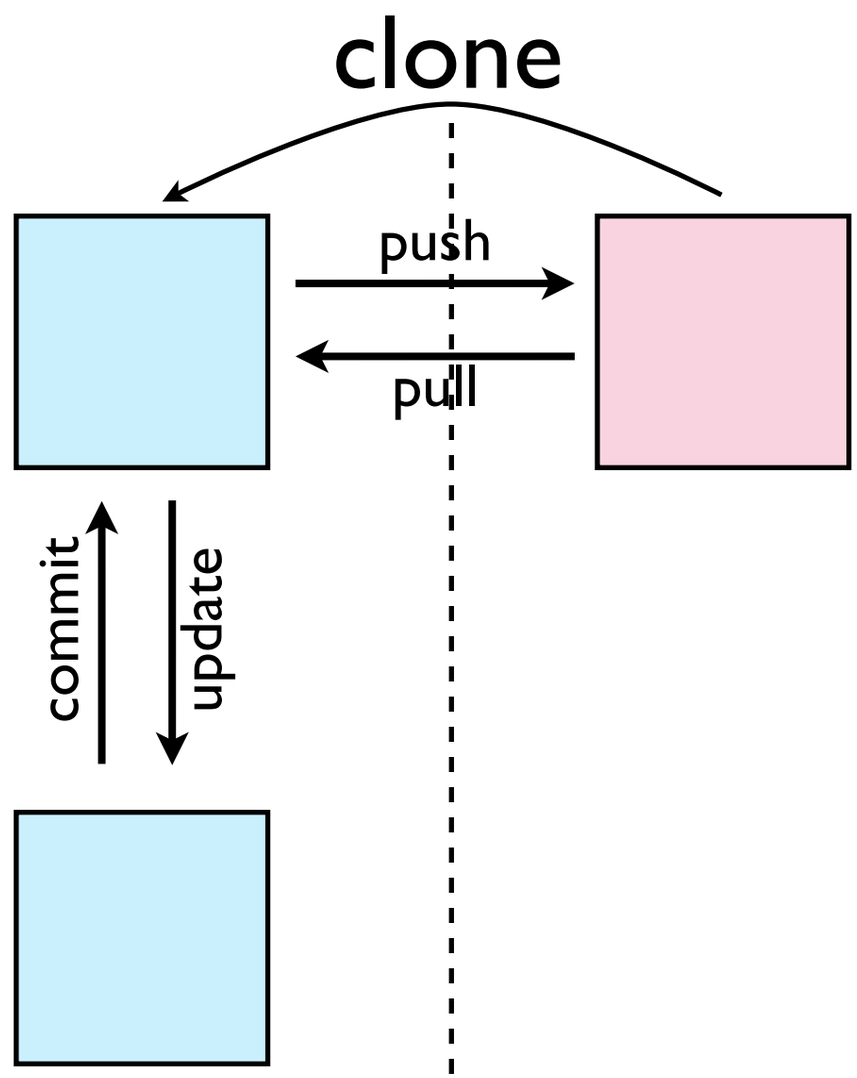
your computer

server

Mercurial: The model

repositories

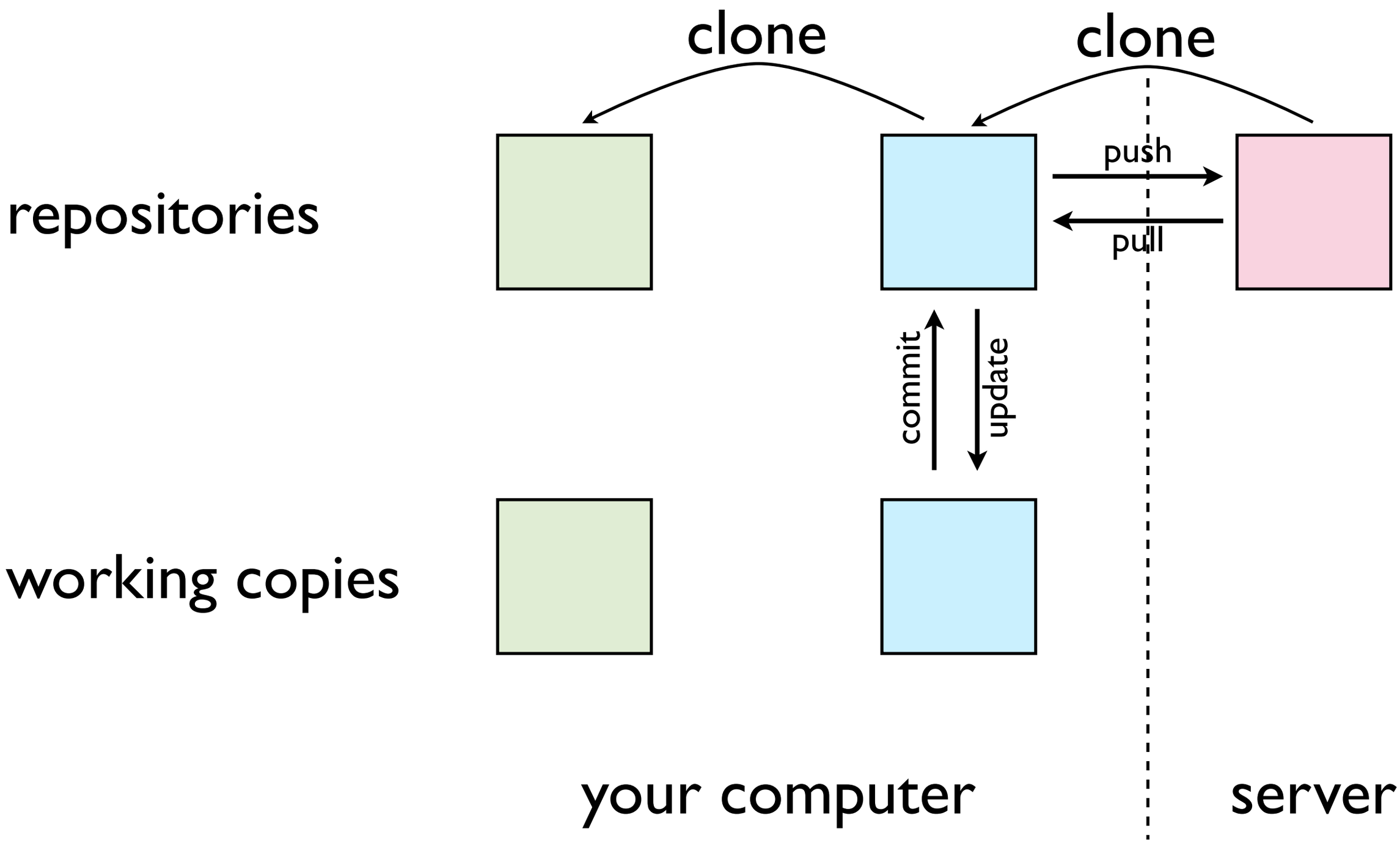
working copies



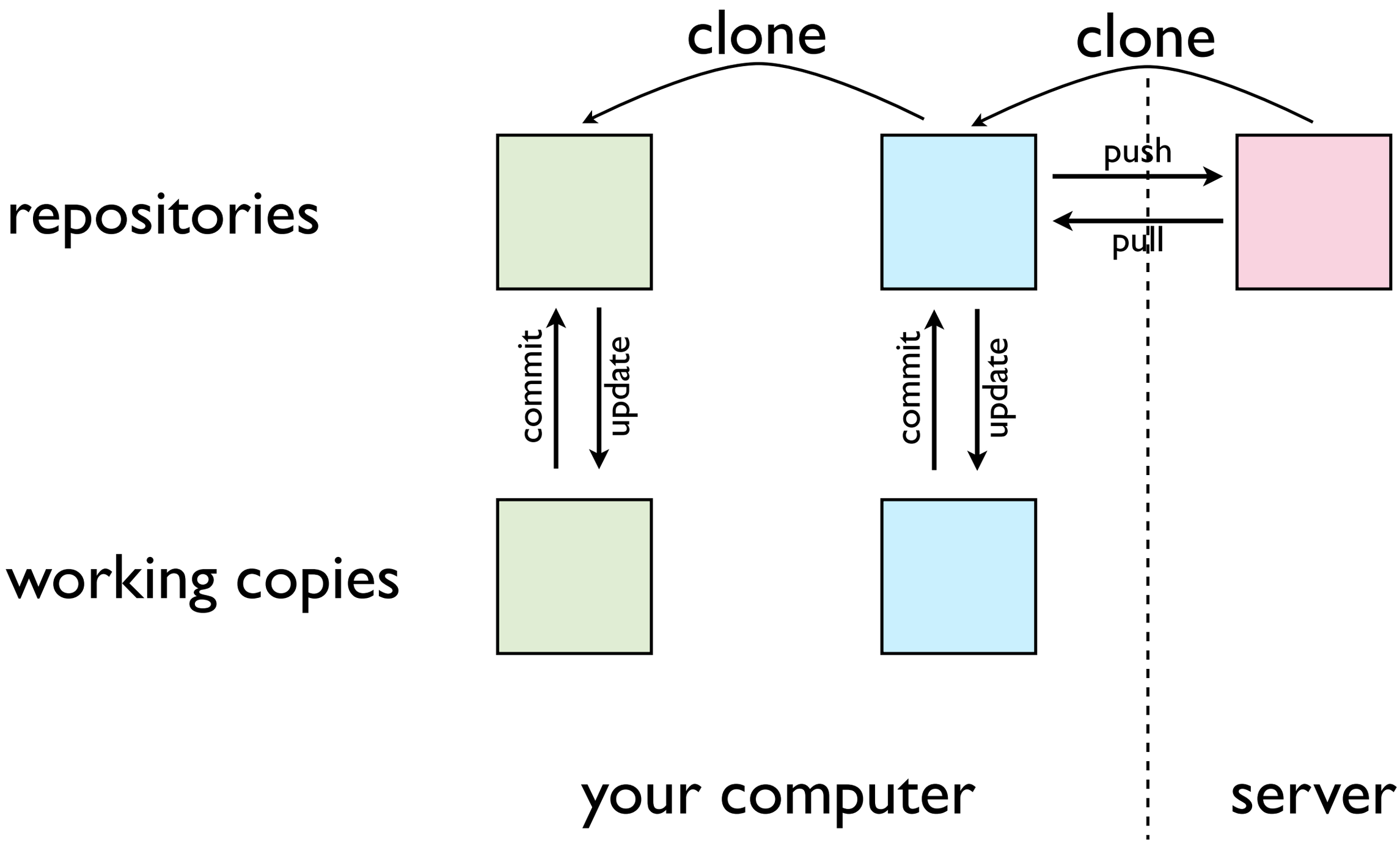
your computer

server

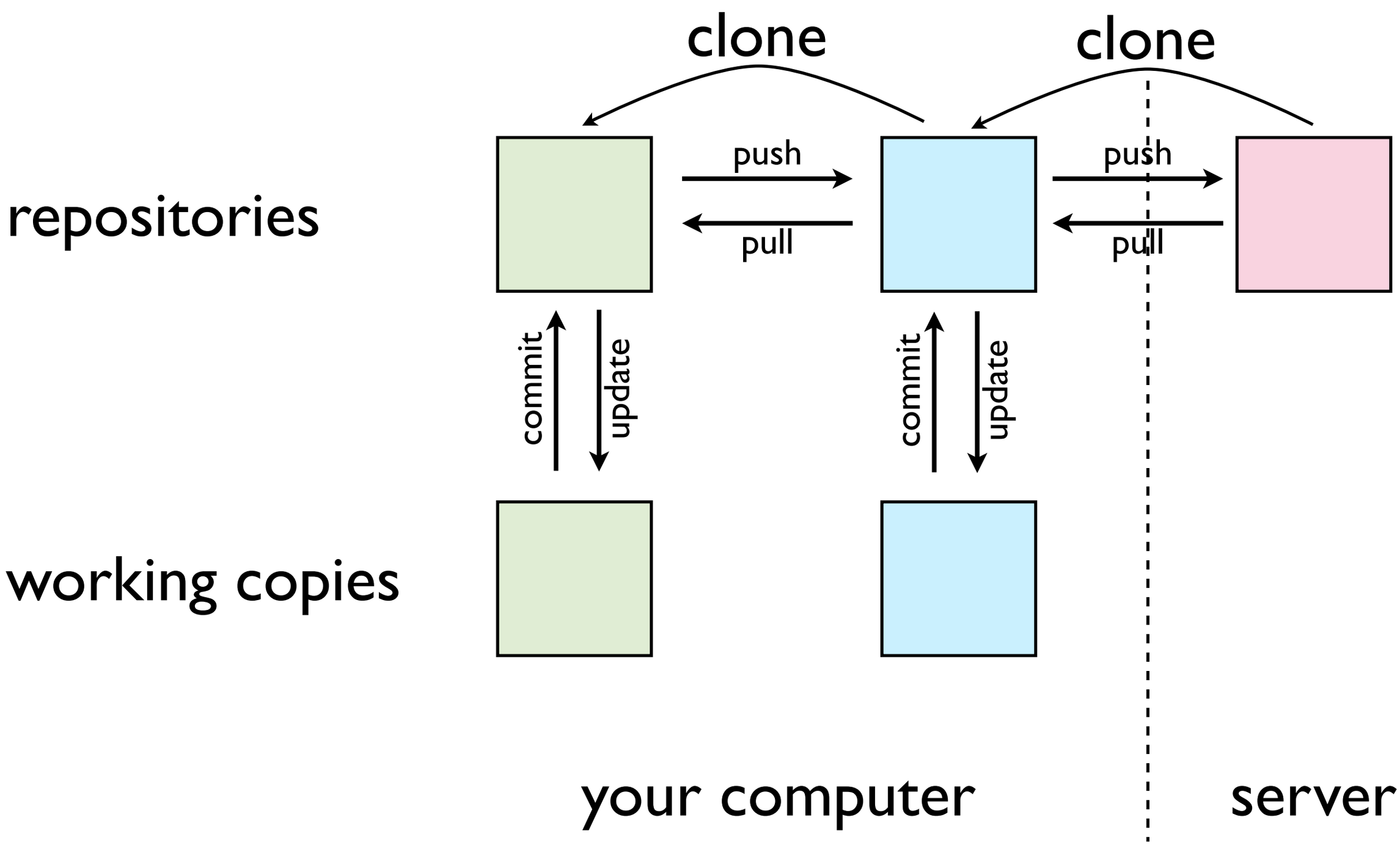
Mercurial: The model



Mercurial: The model



Mercurial: The model



Learning about Mercurial

- Mercurial can be a bit tricky to understand, but there are tons of tutorials on the Web.

- Try it out!

Other tools

- Issue tracking software:
  - database of bugs and improvements that the team wants to work on
  - ideally, each issue (or "case" in FogBugz) assigned to one developer
  - with estimate of how much time it will take
- Discussion groups and Wiki.
- Start using them as much as you like!

Maintaining code quality

- Unit tests:
  - small bits of code that test whether small pieces of your program work individually
  - see e.g. `http://tinyurl.com/8bneq`

- Code reviews:
  - each major piece of code needs to be approved by some other developer before being added to repository
  - see e.g. `http://tinyurl.com/yk7spmt`

Managing the software project

- I'd like to try out the Scrum software project management approach: http://en.wikipedia.org/wiki/Scrum_(development)

- You're the team; each of you gets to be Scrum Master for one sprint; I'm the Product Owner.

- I encourage you to try collaboration tools like testing and reviewing as needed.

Homework (I)

- Write a tiny GIVE system of your own.
  - it needs to do something simple that goes beyond the dummy NLG system
  - it needs to contain at least one new class
  - it needs to compile with "mvn install assembly:assembly"
  - add it to the central Mercurial repository
  - email me where it is by Wednesday 4pm, so I can try it out before the next class

- You can use any programming language you like. Be aware of Java, Scala, Groovy, Jython.

Homework (II)

- Learn more about collaboration tools.
  - create and resolve a case in FogBugz
  - team up with a friend and play around with the code review tool
  - write a simple class with a unit test, and get Maven to run the unit test
  - post in the FogBugz discussion group and/or wiki
- Our FogBugz site is there for you; feel free to explore and use it!