UNIVERSITATEA TEHNICĂ A MOLDOVEI

Network Programming

Cristofor Fiștic

Autumn 2026

Course Description

PR

Concurrency primitives + protocols

PTR

Concurrency w/ messages + streaming

PAD

Distributed systems and their perils

- **Topics** - first Concurrency, and then Protocols, also a bit about networks and Git
- **Labs** - 4x, based on main topics
- **Midterms** - two midterms, a lab + questions (3 Qs)
- **Exam** - oral, interview, Q&A (**Unless I decide otherwise)**
- **Attendance** - Matter. “Decant” will verify the number of absence. I will appreciate your presence.

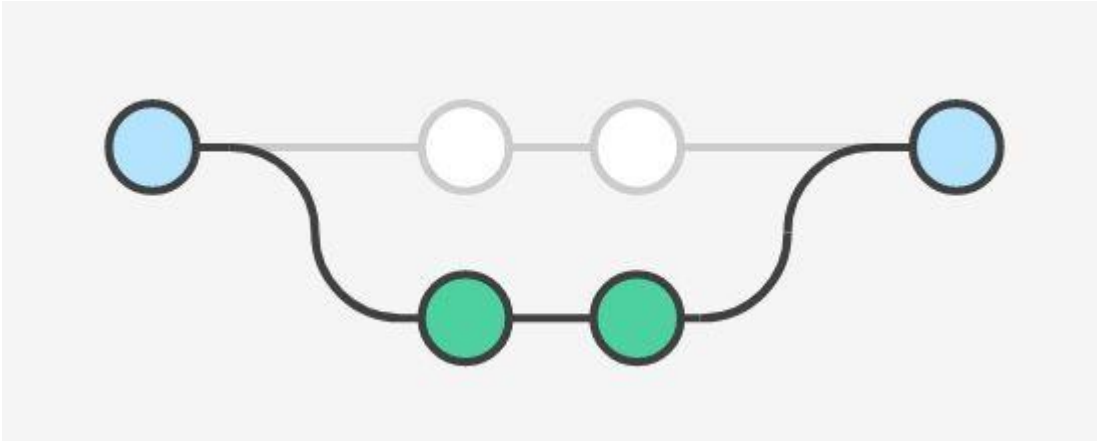
A. Git
B. Computer Networks
C. Communication
D. Concurrency
E. Protocols

- 

Let’s talk about Git

What is Git ?

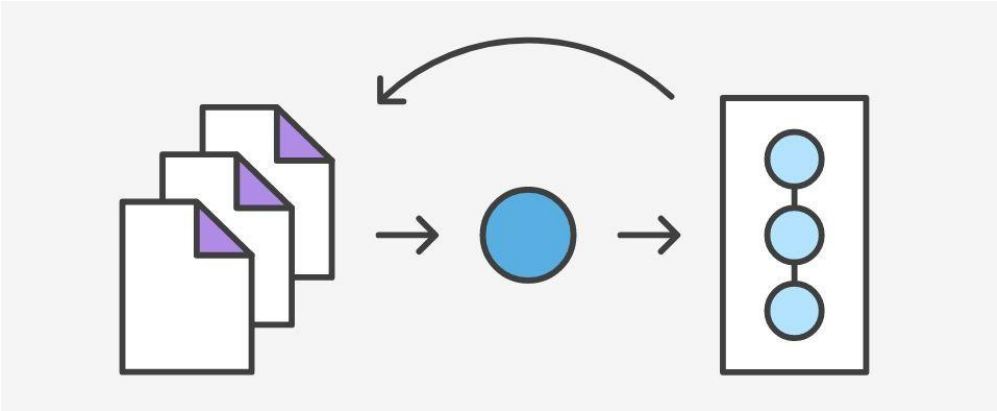
Quick Check 1



Merge vs Rebase?

*Img. Source: https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Quick Check 2



Go back to previous state?

*Img. Source: https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting

Quick Check 3, 4, 5

- Git revert vs reset ?
- Git fetch vs pull ?
- Git tag vs branch ?

Quick Check 6, 7, 8

- How’s SVN different from Git?
- Git stash anyone?
- Git Flow? Forking workflow?

Must know on Git internals

Git is indeed an interesting VCS. It's author claims that he was heavily inspired by file systems rather than other VCS, that's why Git is so different.

Git stores *snapshots* of the working directory. This allows fast checkout between commits and branches. But this consumes **a lot of memory**.

The Core Objects:

- **Blob (Binary Large Object):** The content of a file.
- **Tree:** A directory listing, containing pointers to blobs and other trees.
- **Commit:** A snapshot pointer to a single top-level tree. It also contains metadata (author, message) and pointers to parent commits, forming the project history

Must know on Git internals

That's why sometimes when git does garbage collection, or is forced to (**git gc**) it will compress a number of snapshots into packfiles and It cleans up unnecessary objects. Git runs this automatically when it thinks it's needed.

Now, you really should check the Git book: https://git-scm.com/book/en/v2, especially 10th chapter.

Must know on Git internals

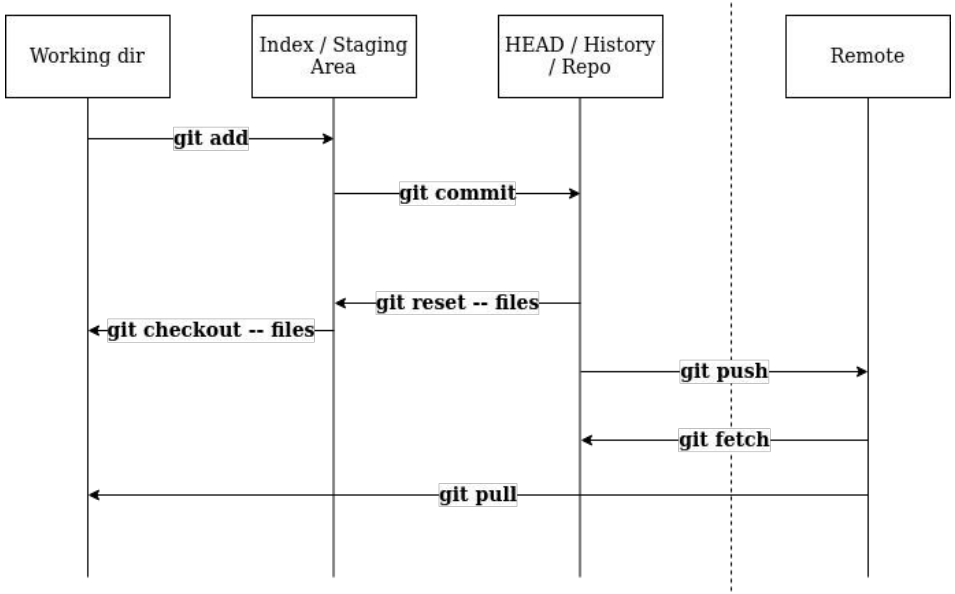
What is a Packfile? During garbage collection, Git takes all the individual objects and bundles them into a single, highly compressed binary file called a **packfile**.

How does it work? Instead of storing hundreds of full snapshots, Git finds similar files. It stores one version as a whole and then saves all the other versions as just the **deltas** (the differences) from that base file.

Why it Matters: This delta compression is incredibly efficient. It drastically reduces the size of your repository, making cloning, fetching, and pushing much faster, especially for large projects with long histories.

Must know on Git internals #2

For more info with nice visualizations, check: marklodato.github.io/visual-git-guide/index-en.html



Some cool things that Git does

Some cool things that Git does

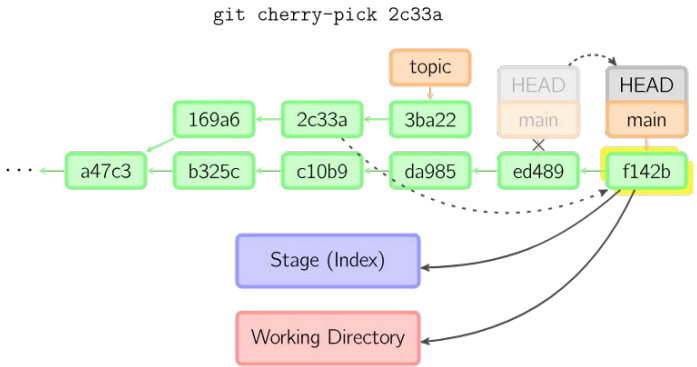
- **git bisect** If you know a bug was introduced somewhere between a "good" old commit and the current "bad" commit. It performs an automated binary search through your commit history. You simply tell it a known good commit and a known bad one, and Git will check out commits in between, asking you to test each one. This process quickly narrows down the exact commit that introduced the bug.

How to use it:

1. git bisect start
2. git bisect bad *(marks current commit as broken)*
3. git bisect good <hash_of_working_commit>
4. Git checkout a commit. Test it, then run git bisect good or git bisect bad.
5. Repeat until Git points out the culprit!
6. git bisect reset

Some very cool things Git does

- **git cherry-pick** What if you solved a bug in a branch, and want to propagate the change to everyone else? Doing a full fledged merge could cause anything from a lot of downtime solving merge conflicts and messy commit history, to even losing credibility and respect at your workplace. Don't worry, **git cherry-pick** got you covered! Now you can pick a single commit and propagate your bugfix or whatever.



Some very cool things Git does

- **git submodule** A feature that lets you keep a separate Git repository in a subdirectory of your main repository. It's like having a project within a project. Perfect for managing external dependencies, shared libraries, or components developed in another repo. You can use a specific version of the other project without copying its code into your main project.

How to use it:

1. **Add a submodule:** git submodule add <repository_url> path/to/submodule
2. **Clone a project with submodules:** git clone --recurse-submodules <main_project_url>
3. **Update submodules to the latest commit:** git submodule update --remote

Some very cool things Git does

- **git blame** When you encounter a confusing line of code and need to ask someone for clarification, git blame is your tool. It annotates each line in a file with the commit hash, author, and timestamp of the last modification. This isn't for blaming people, but for **finding context** and the right person to talk to. Most IDEs, like VS Code (GitLens extension) and JetBrains, have this functionality built-in.

How to use it:

- git blame <file_name>

Some very cool things Git does

- **git worktree** Normally, you can only have one branch checked out at a time in your repository. Git worktree lets you check out multiple branches into separate directories linked to the same repository. This is perfect when you need to work on a quick hotfix in main without stashing or committing your in-progress work on a feature branch

Some very cool things Git does

You know git rebase, but the interactive mode (-i) is a game-changer for cleaning up your local commit history before creating a pull request. It opens an editor with a list of your commits and allows you to:

- **reword**: Change the commit message.
- **edit**: Amend the commit's content.
- **squash**: Combine multiple commits into one.
- **fixup**: Like squash, but discards the commit's log message.
- **drop**: Delete a commit entirely.
- **reorder**: Change the order of commits.

A clean, logical commit history makes your code much easier to review and understand.

How to use it:

- git rebase -i HEAD~5 (to interactively rebase the last 5 commits)

Some very cool things Git does

Many beginners see the staging area as just a hurdle before committing. But it's a powerful tool for crafting **atomic commits**—commits that contain a single, complete logical change. Instead of adding all your changes with git add ., you can stage specific parts of a file. This lets you separate unrelated changes (e.g., a bug fix and a new feature) from the same file into different commits.

How to use it:

- git add -p or git add --patch
- This command will go through your changes hunk by hunk, asking if you want to stage each one.

Some very cool things Git does

git stash is great for saving uncommitted changes. But did you know you can manage multiple stashes?

- git stash save "a descriptive message": Stashes changes with a message so you remember what it was.
- git stash list: Shows you all the stashes you've saved.
- git stash apply stash@{2}: Applies a specific stash from the list without dropping it.
- git stash pop stash@{2}: Applies a specific stash and then drops it from the list.
- git stash show -p stash@{0}: Shows the changes in a specific stash as a patch.

Some very cool things Git does

- **Git LFS (Large File Storage)** An extension for Git that replaces large files (like audio, video, datasets, or graphics) with small text pointers inside your repository. The actual large files are stored on a separate server. Git is not designed to handle large binary files. Committing them directly will bloat your repository, making it slow to clone and work with. Git LFS is the industry-standard solution to this problem.

How it works:

1. git lfs install *(run once per machine)*
2. git lfs track "*.psd" *(tells LFS to handle all Photoshop files)*
3. Commit the .gitattributes file that was created.
4. From then on, just git add and git commit as usual. LFS handles the rest automatically.

Some very cool things Git does

- **git daemon** A simple, built-in server that turns your repository into a public, read-only resource on your local network. Perfect for quickly sharing code with colleagues on the same network without needing to set up SSH keys(Hackathons), push to a remote server like GitHub, or configure complex permissions. It's unauthenticated and lightweight.

How to use it:

1. Navigate into your Git repository's main directory (.git folder).
2. Run the command: git daemon --base-path=. --export-all
3. Others can now clone your repo using: git clone git://<your-local-ip-address>/

Some very cool things Git does

Git hooks are scripts that run automatically at certain points in the Git lifecycle, like before a commit (pre-commit) or before a push (pre-push). They are stored in the .git/hooks directory. You can use them to:

- Lint your code before it's committed.
- Run unit tests before you push.
- Check if commit messages follow a specific format.

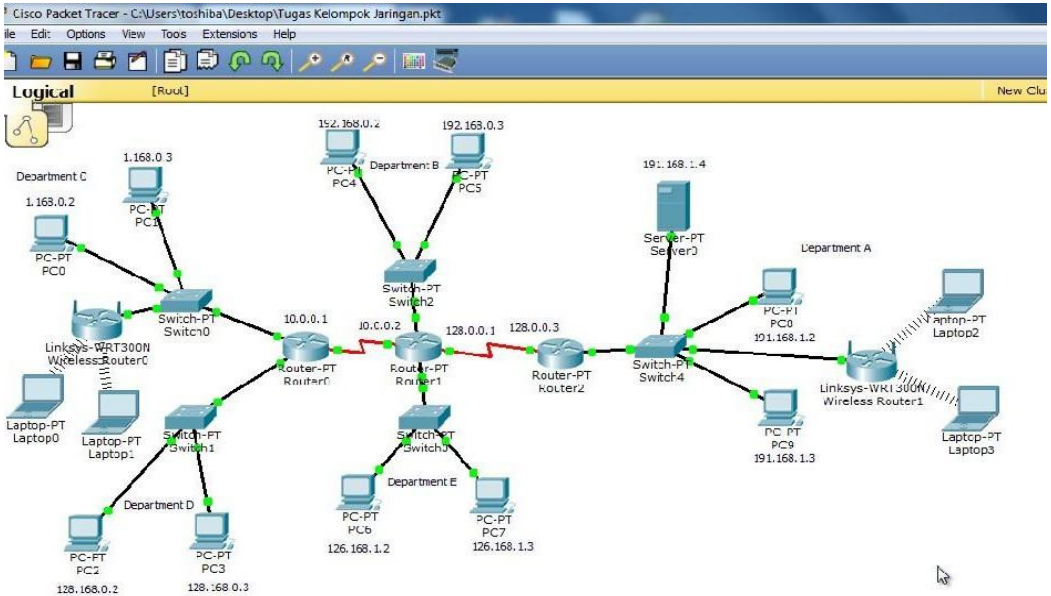
This helps maintain code quality and prevent broken code from ever reaching the remote repository. Tools like **Husky** make managing Git hooks much easier.

Reading list

- http://tom.preston-werner.com/2009/05/19/the-git-parable.html
- https://nvie.com/posts/a-successful-git-branching-model/
- https://www.endoflineblog.com/gitflow-considered-harmful
- https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow
- https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow
- [For fanatics and no-lifers] Merkle Tree. https://www.codeproject.com/Articles/1176140/Understanding-Merkle-Trees-Why-use-them-who-uses-t

How it all began...

"There was an idea To bring together
A group of remarkable machines So when we need them
They could compute the stuff That we never could
... and also survive a couple of nukes" **- Nick Fury**



Computer networks

Ok, now seriously

Among the first computer networks where established by U.S. military in late 50s, notoriously SAGE aka Semi-automatic Ground Environment.
Another important milestone was SABRE, a commercial airline booking system, made in 1960, which was two mainframes connected.

The network that you probably all know about was 1669's ARPANET, the precursor of The Internet, which was connecting a number of research institutions and military facilities, financed by DoD, and having the main objective to survive a nuclear war.

What made it so innovative was the packet switching technology, the possibility to access the data and applications on any of the machines within the network from any other machine, also it had the first email.

Packet Switching. Why? How?

What makes modern computer networks so efficient is the packet switching technology.

It is used to ensure maximum chance of delivery, with latency and speed as good as possible.

Basically, that means that information is splitted into packets of predefined size and sent over network. In case there's some congestion on the way, the packets are switched through other routes, thankfully the networks have some redundancy.

Packets are sent through the fastest route, but that doesn't always mean it is the shortest route. Think GPS with information about traffic jams.

Packet Switching. Why? How?

A packet is sort of a wrapper around the chunk of data that the system is trying to send.
It has a header and a trailer. Most meta information is in the header, while the trailer is used to make clear how where the packet is ending.

For example, the header of a packet contains the sender and receiver addresses, the number of packets to be sent, the id of a specific packet, some control information.

For TCP, if a packet is lost or corrupted, it is resent, but more on that later.

A quick refresher: Topology

First there was the **the mainframe** with multiprogramming capabilities, and many client machines in a **star** topology.

Then, came **the bus** and later **the ring** topologies. But it was never enough.

And later, the router was invented and compute machines became affordable. At that point networks could be built just like they are now, more or less. So the **mesh** topology emerged. Relatively recently peer-to-peer (P2P) networks emerged.

And there are also ad-hoc networks, like P2P but ad-hoc.

A quick refresher 2: Computer networks challenges

More computers, that are also interlinked, plus having devices to link them and provide different services, all this makes the system more fragile/prone to issues and vulnerabilities.

1. High maintenance cost
2. Increased budget for devices and software
3. Necessity for specialized staff to keep the network running normally
4. Security concerns (higher attack surface for hackers, spread of malware)
5. Authorization and access control
6. Basically your machines could burn, so you need backups
7. Et cetera

A quick refresher 3: Computer networks requirements

For a computer network to be useful, it should be analyzed, and perform well in certain dimensions.

1. Functionality - the more, the merrier
2. Efficiency (latency, response time, transfer speed, bandwidth)
3. Resilience and fault tolerance
4. Security
5. Availability
6. Quality of Service
7. Integrity and coherence
8. Monitoring and traffic control
9. Scalability
10. Adaptability

Computer networks standards

To be extensible and scalable… and easy and cheap to maintain, … and interoperable, the components of a network are desirable to be obey **Open Standards**. Both **Open** and **Standards** are important words here. Let's dissect.

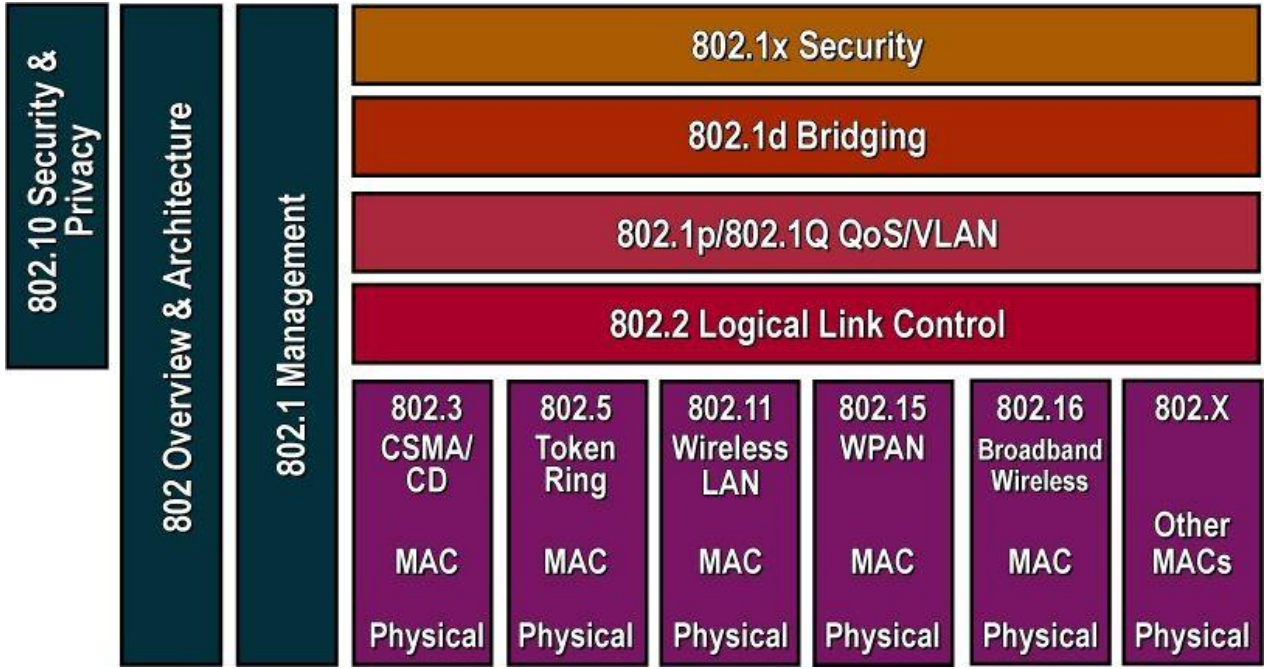
Open - open means that the specification is freely available for everyone, and doesn't have licencing costs. Specifications are not standards.

Standards - standards are complete, formal specifications that are crystalized and through a consensus decided to be canonical. Standards are maintained and developed by standardization institutions, like ISO, ANSI, IEEE, NIST, ECMA, or others.

The IEEE 802 project

Since February 1980, IEEE started working on
802.x suite of standards that cover all network components.

Just wikipedia it.



*Source: www.hill2dot0.com/wiki/index.php?title=IEEE_Project_802

Internet development organizations

ISOC (INTERNET Society) - Internet evolution and development. **IAB (Internet Architecture Board)** - technical control. **IETF (Internet Engineering Task Force) and IRTF (Internet Research Task Force)** - solve actual problems and do long term research, correspondingly.

Communication

Communication objectives

We have 3 primary reasons to use communication:

- Data transfer, as in HPC (Accelerate computing), or microservices (Modularization)
- Data sharing, as in Google Drive, or Amazon S3
- Synchronization, or control, as in telling the other party what to do

*Source: Communication Over Networking - PR 2017, UTM, conf.univ. PhD Ciorbă Dumitru

Communication paradigms

Shared memory

- Efficient
- Error prone
- Really good when processes are local to a machine

Message passing

- Shines when running on multiple machines
- Abstracts away the location of the process
- Less error prone

Remote invocation

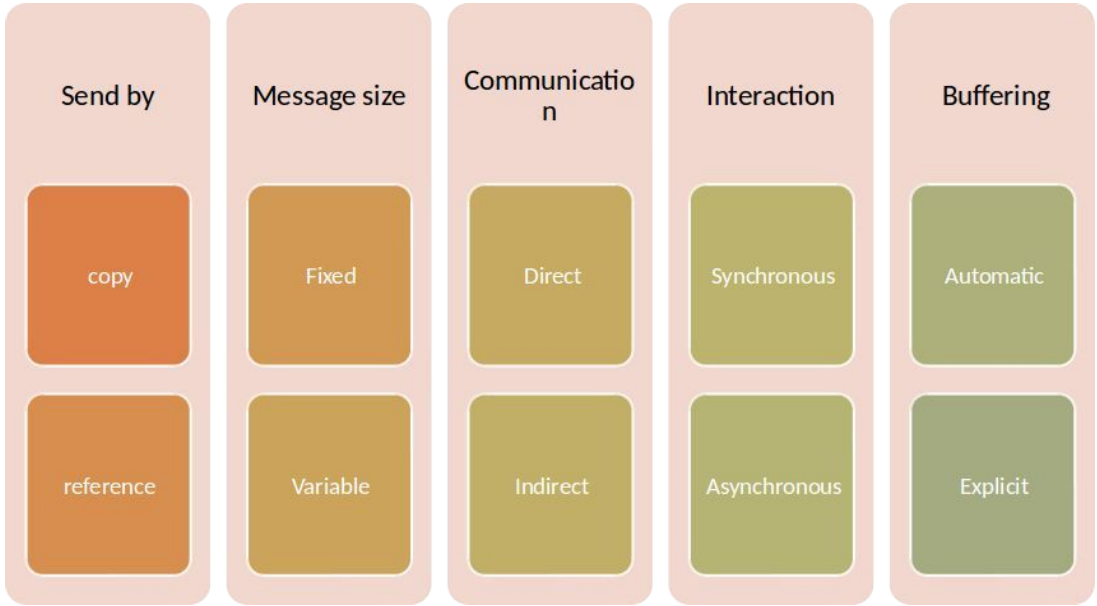
- Just as Message Passing, but now you abstract away even the fact that you have another process. It feels like calling a function

Communication paradigms: Direct and Indirect

Direct Communication - processes communicate directly, obviously, it is efficient but not scalable. And not that flexible too. RPC and Queues and concurrent objects communicate like this.

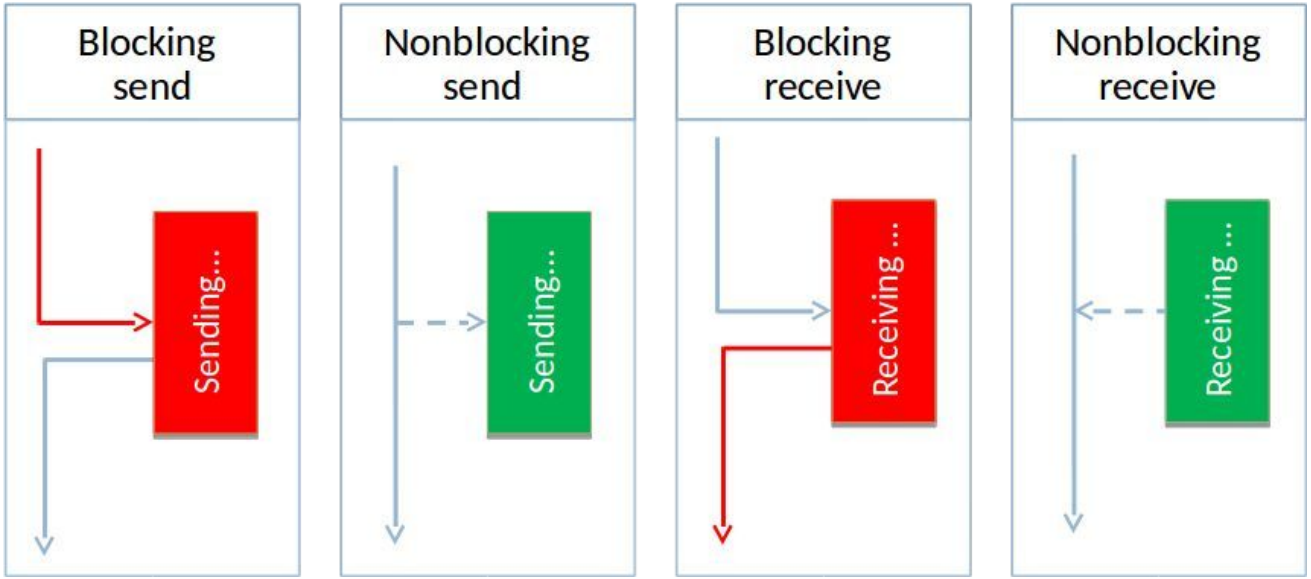
Indirect Communication - processes communicate through an intermediary, like a message queue or a mailbox. It is less efficient but lets you build systems which can handle variable number of agents communicating, efficiently.

Data Transfer. Dimensions.



*Source: Communication Over Networking - PR 2017, UTM, conf.univ. PhD Ciorbă Dumitru

Data Transfer. Interactions.



*Source: Communication Over Networking - PR 2017, UTM, conf.univ. PhD Ciorbă Dumitru

Data Transfer. Buffering.

Capacity	Buffering procedure

• Zero
  • ~~Asynchronous~~
  • Synchronous
• Bounded
• Unbounded
	• Automatic
	• Explicit

*Source: Communication Over Networking - PR 2017, UTM, conf.univ. PhD Ciorbă Dumitru

Communication granularity

Services - REST or SOAP or RPCs (Java RMI, .NET Remoting, CORBA, gRPC, Thrift) and all that network protocols of high level

Messages - Message Queues, WebSockets, Events and pipes/streams

Bytes - Files and pipes, Sockets (BSD ones)

Pipes

Pipes are an inter-process communication method (IPC) which shares some concepts with both shared memory and message passing paradigms.

On implementation level pipes are special temporary files. Pipes obey the so-called **stream processing** paradigm, which says that there's only one concrete producer and consumer and that the communication is unidirectional.

Pipes are usually OS primitives, and sometimes one can be bidirectional and named.

Pipes

The simplest example of a pipe, in UNIX systems, would be something like this expression `ps -aux | grep -E "<some regexp pattern>"` Do you know what it does?

The `|` operator creates a temporary store in memory in which the output content from the first command is written into, and simultaneously used as input for the second command. This is an **anonymous pipe**. Pipes are good as IPC within a system. They are usually faster than sockets.

There are also named pipes. Here's how these are created in Linux `mkfifo <pipe_name>`. Having a named pipe is useful because it can be used as a channel between processes and it won't go away when the processes are done. By alternating read and write sessions, these can ensure bidirectional communication, so are kind of Half-duplex (HDX).

See also: https://stackoverflow.com/questions/18568089/whats-the-difference-between-pipes-and-sockets
and http://hassansin.github.io/fun-with-unix-named-pipes
and https://stackoverflow.com/questions/1235958/ipc-performance-named-pipe-vs-socket#1238819

Stating the obvious

Network programming in most cases is about a **client** application/service/whatever talking to a **server**. This is even an architectural pattern, called client-server architecture.

Client

- Wants something from the server
- Establishes the connection

Server

- Fulfils clients desires
- Is oblivious of his clients up until the connection is established
- Can’t normally ask anything from the client